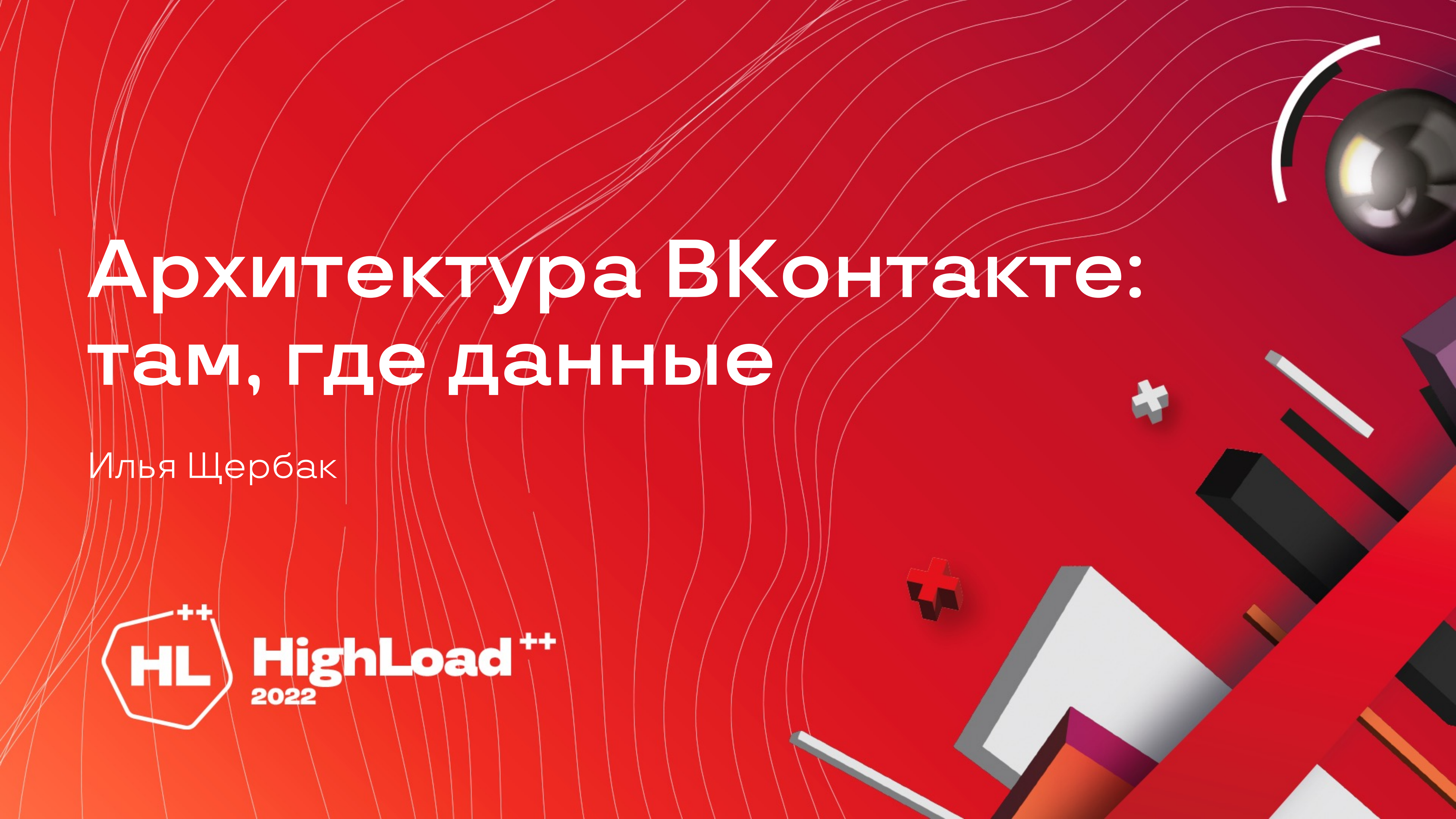


Архитектура ВКонтакте:
там, где данные
Илья Щербак
HL
HighLoad++
2022

О себе



Родился



Учился

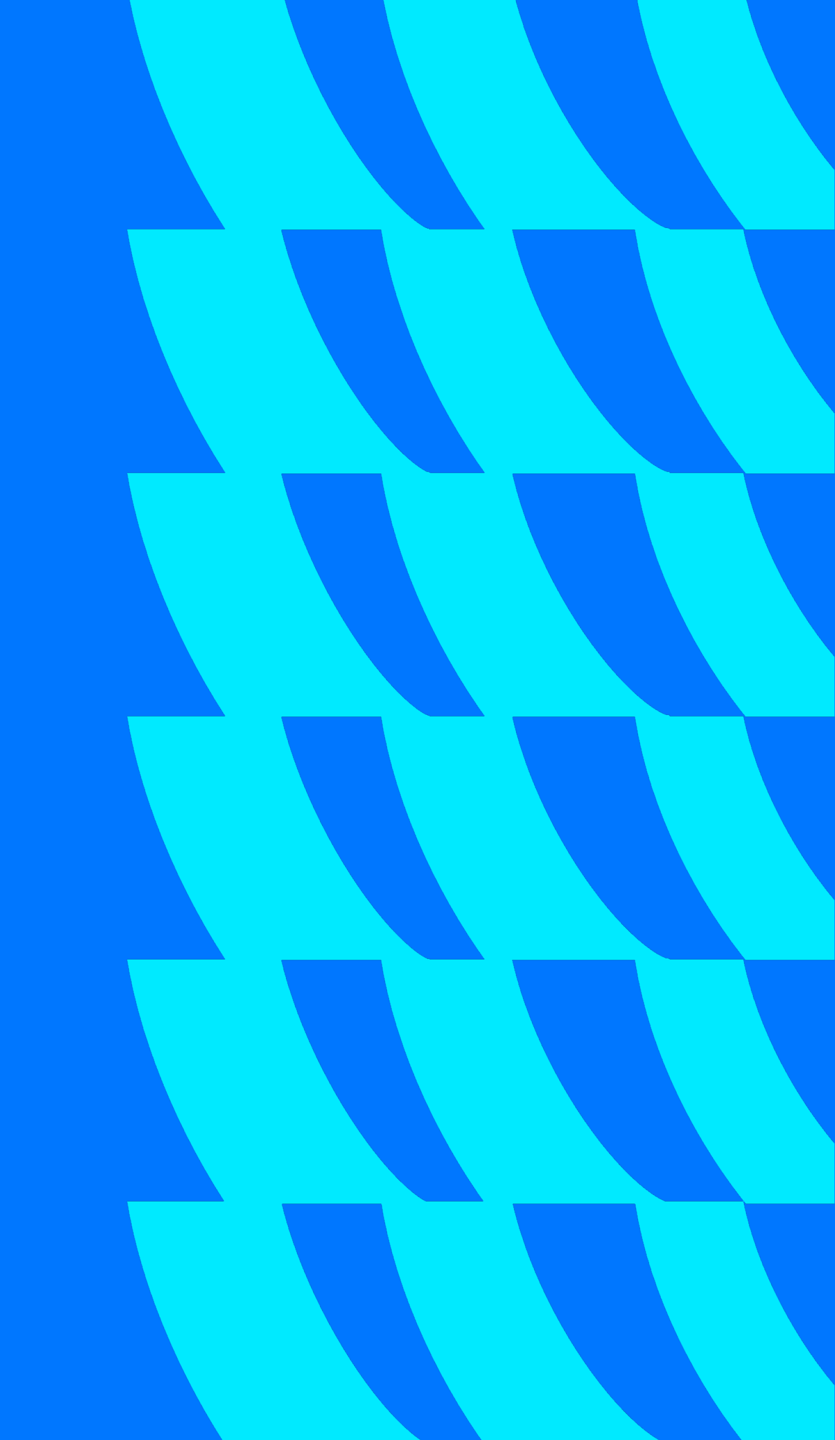


Чему-то научился



Работаю ВКонтакте

Команда
Баз данных
и инфраструктуры
ВКонтакте






Я РАД
ВСЕМ

Технический вектор команды



Отказоустойчивость



Быстродействие



Скорость разработки продукта

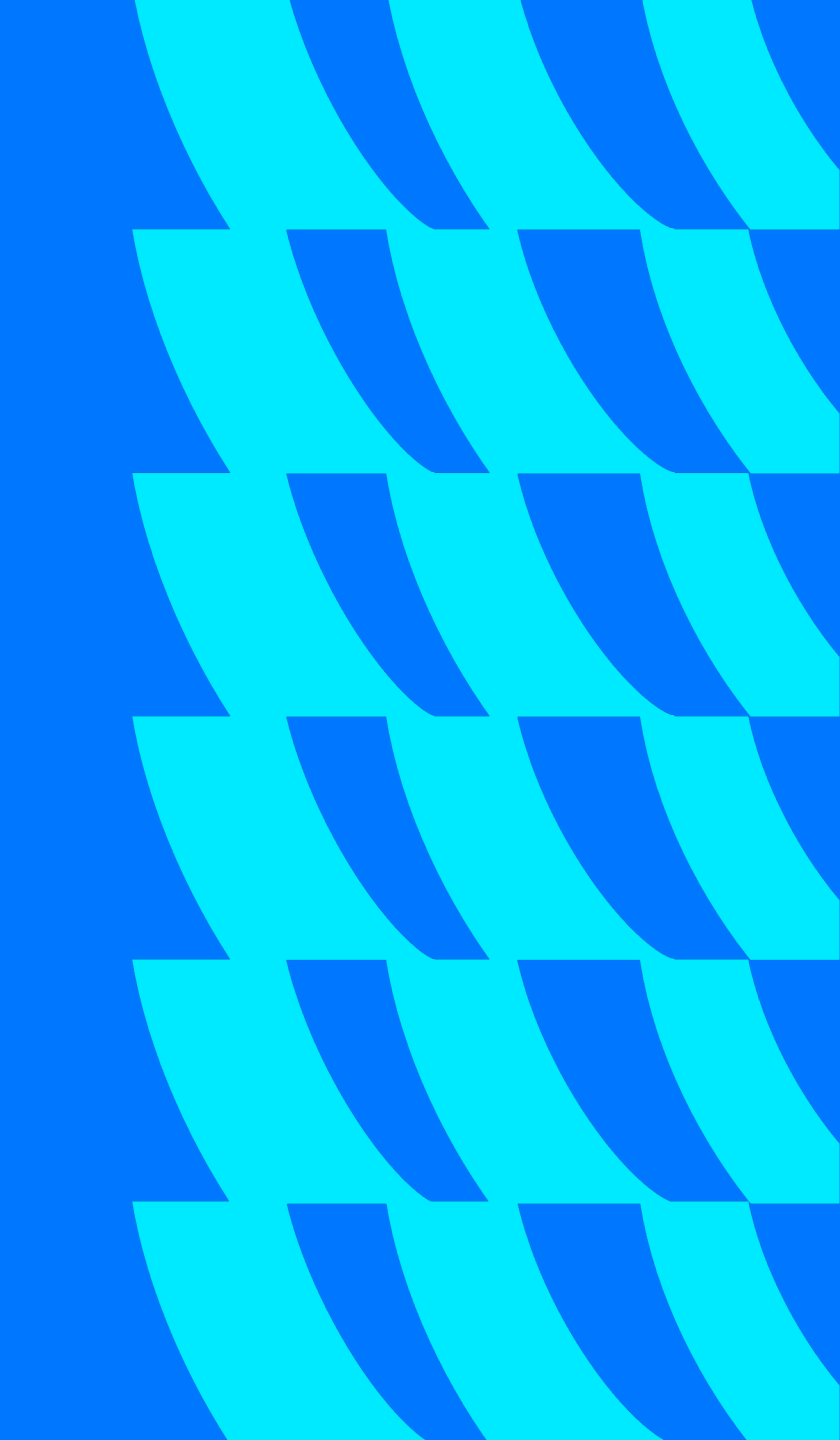


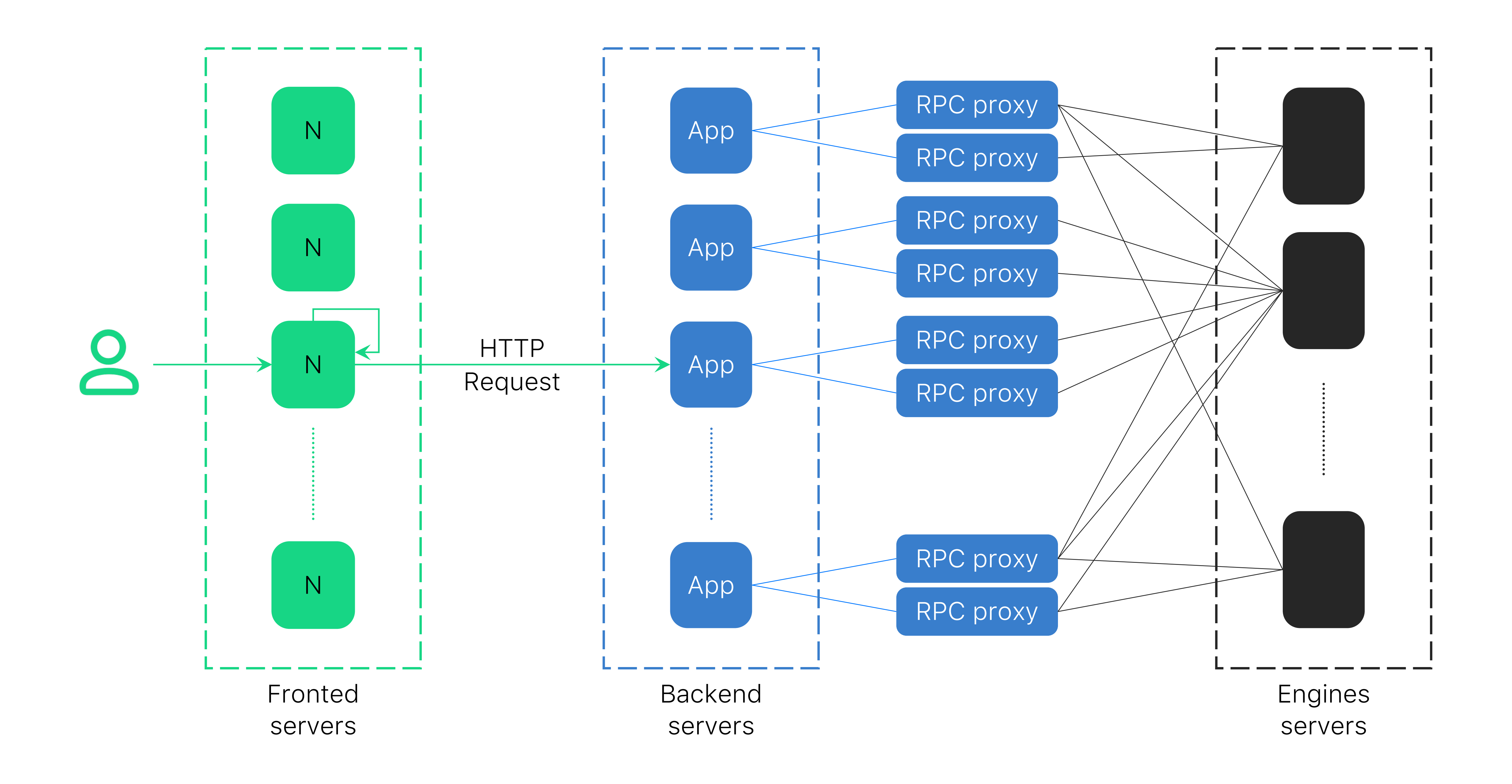
Поддержка продукта



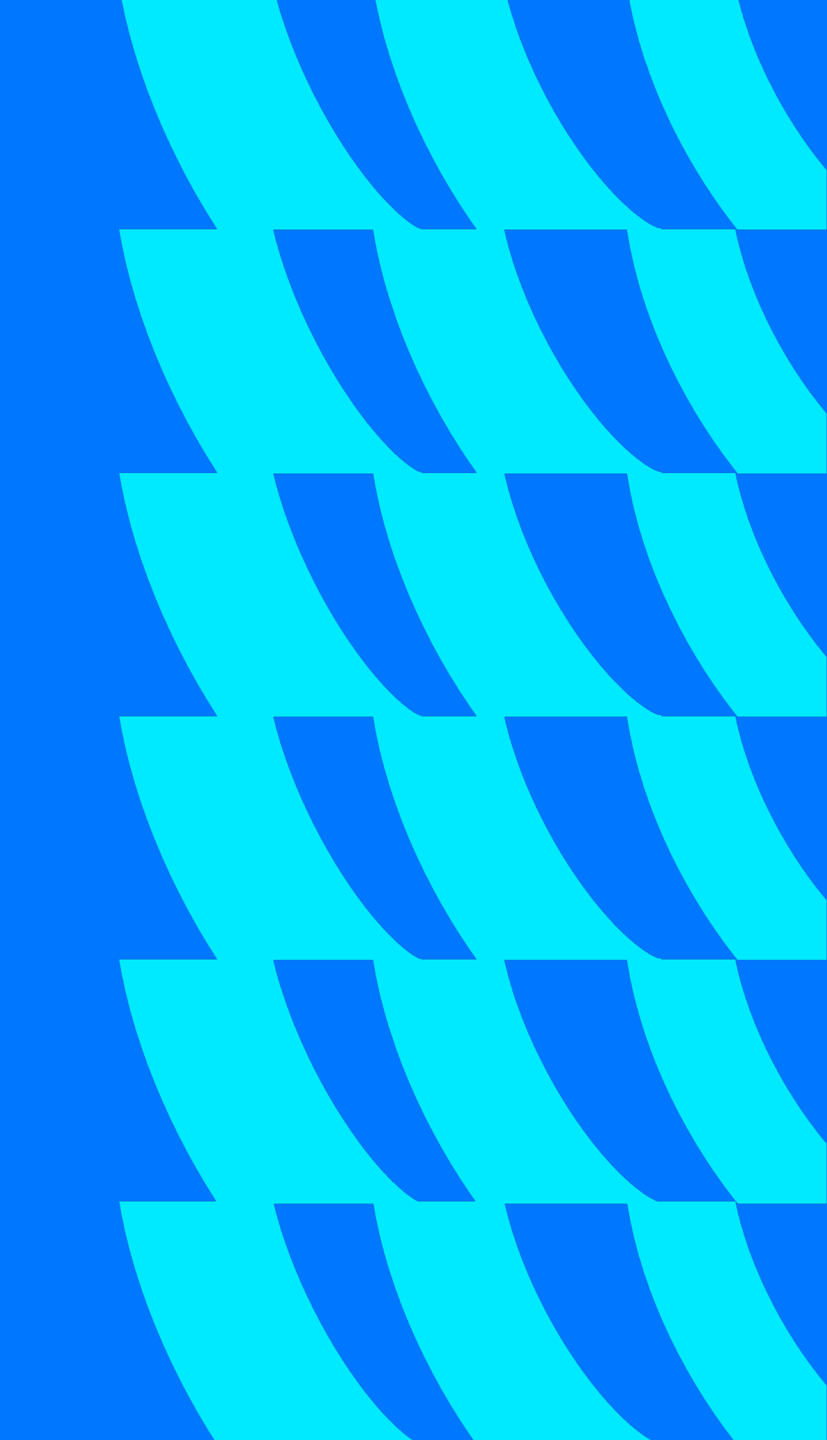
Утилизация ресурсов

Общая архитектура ВКонтакте




N
N
N
N
HTTP
Request
App
App
App
App
RPC proxy
RPC proxy
RPC proxy
RPC proxy
RPC proxy
RPC proxy
RPC proxy
RPC proxy
Fronted
servers
Backend
servers
Engines
servers

Движки ВКонтакте



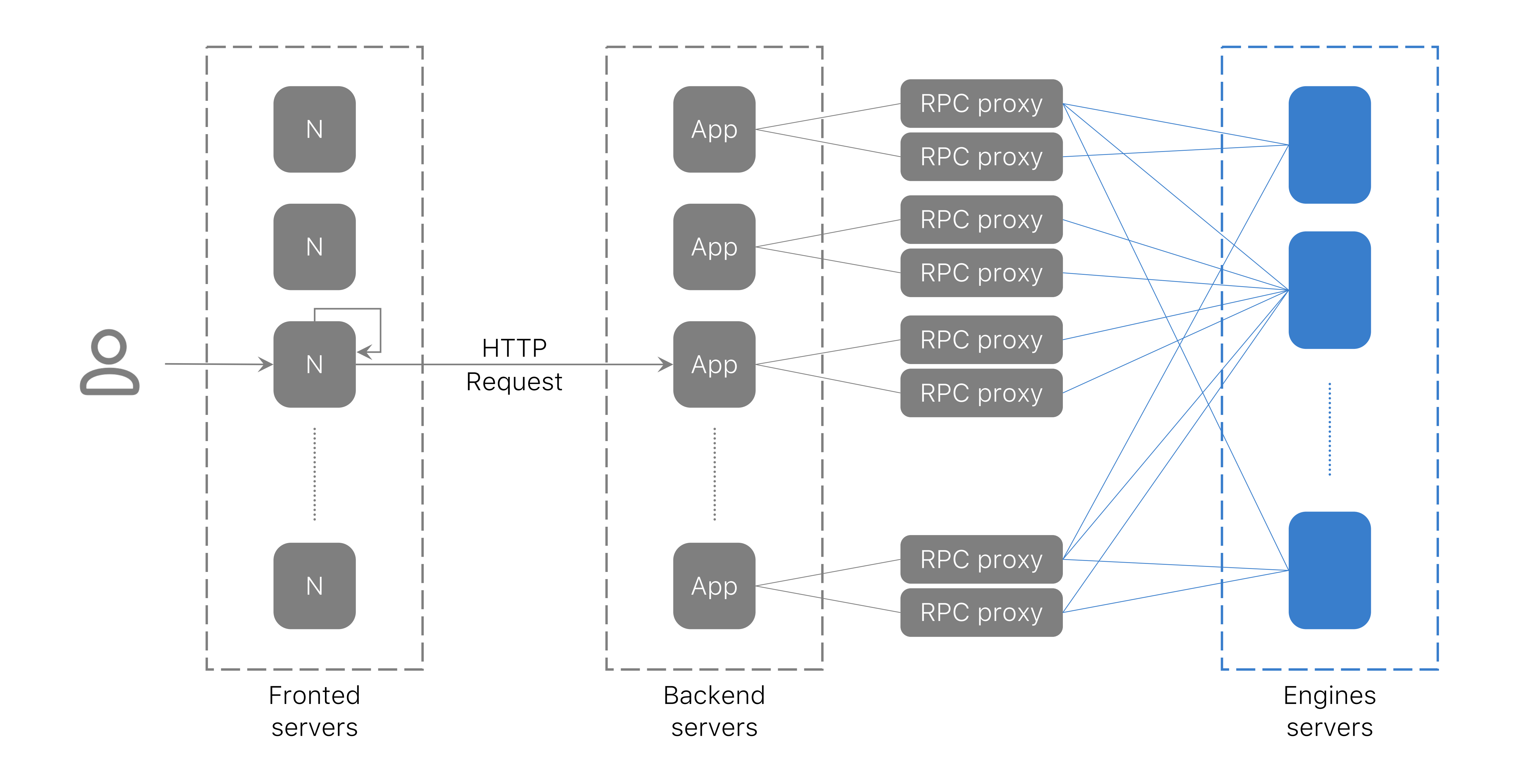
250 mqps

обрабатывают движки запросов в секунду

720 кластеров

кластер — отдельный сетап движка

53 движка


N
N
N
N
HTTP
Request
App
App
App
App
RPC proxy
RPC proxy
RPC proxy
RPC proxy
RPC proxy
RPC proxy
RPC proxy
RPC proxy
Fronted
servers
Backend
servers
Engines
servers

Что делают движки?

1

Хранят и обрабатывают все данные пользователей

2

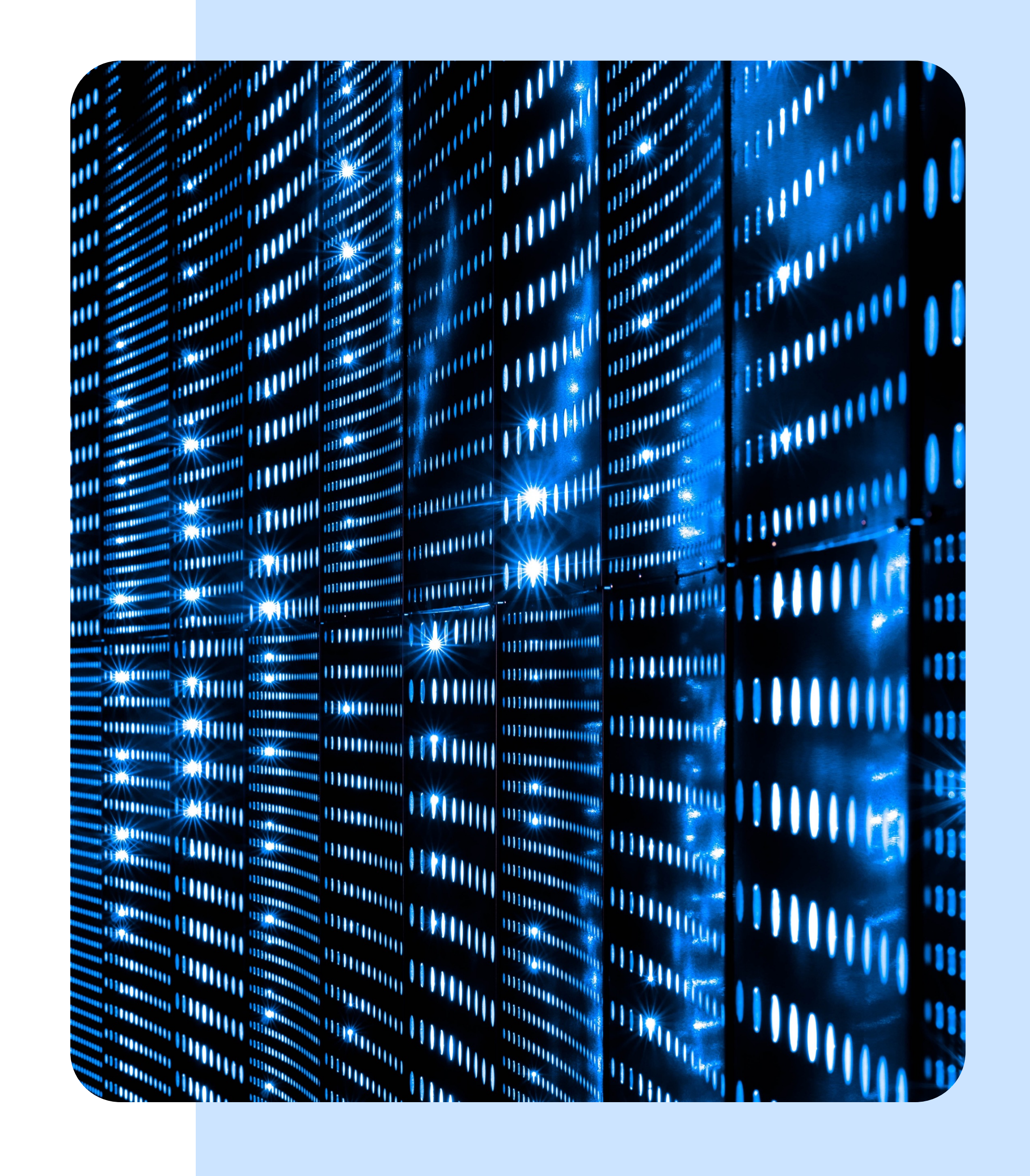
Контролируют и реализуют кеширующий слой

3

Хранят различные служебные данные

4

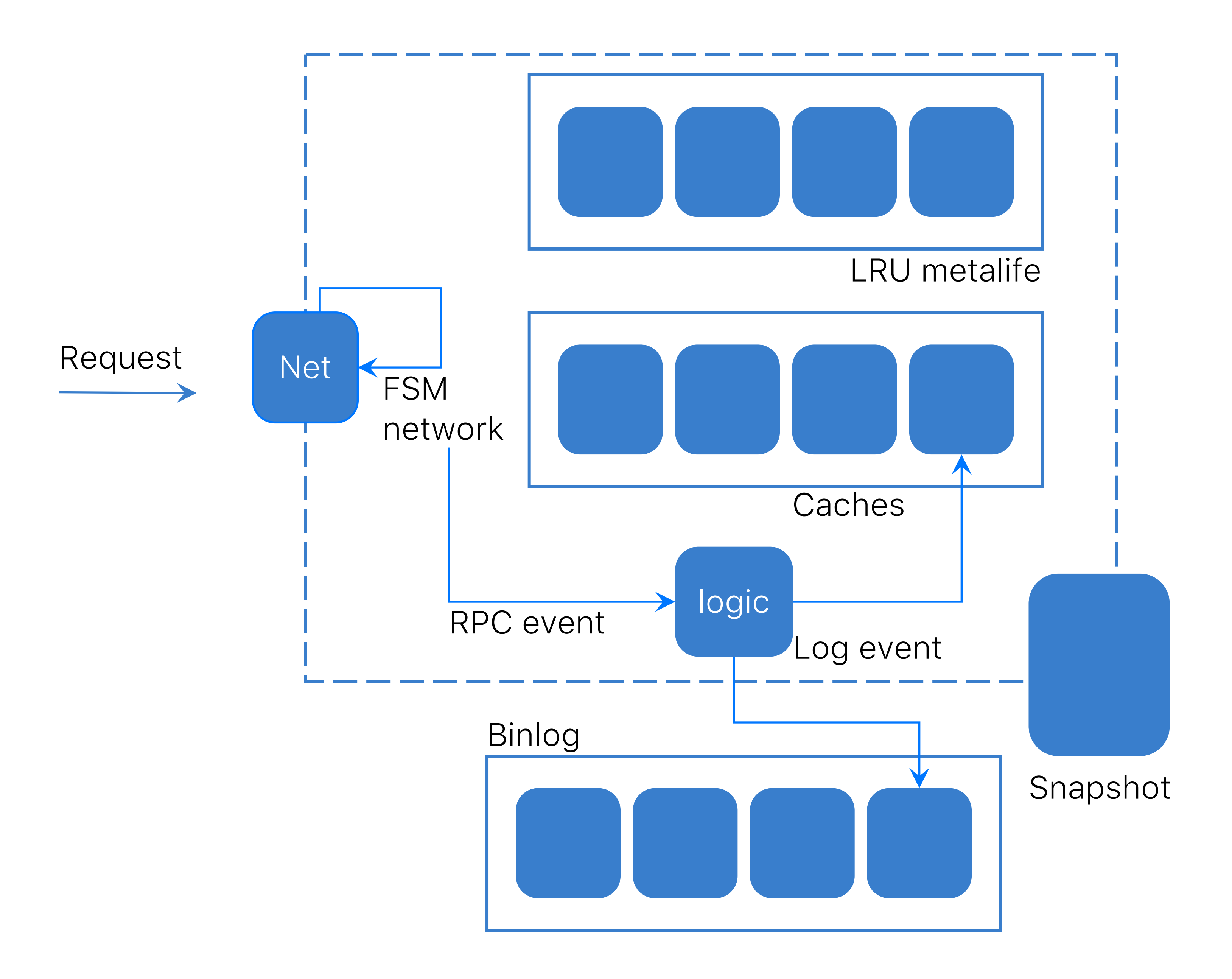
Распространяют конфигурацию



Почему мы РАЗРАБАТЫВАЕМ движки?

1. Гомогенность эксплуатации
2. Экспертность в разработке и поддержке
3. Знание о хранимых данных
4. Общая кодовая база сетевого слоя и работы с диском

Типовой движок



Binlog & snapshot

OP	Users	Value
add	5015	1
add	5030	2
add	5015	2
sub	5030	2
add	5030	5

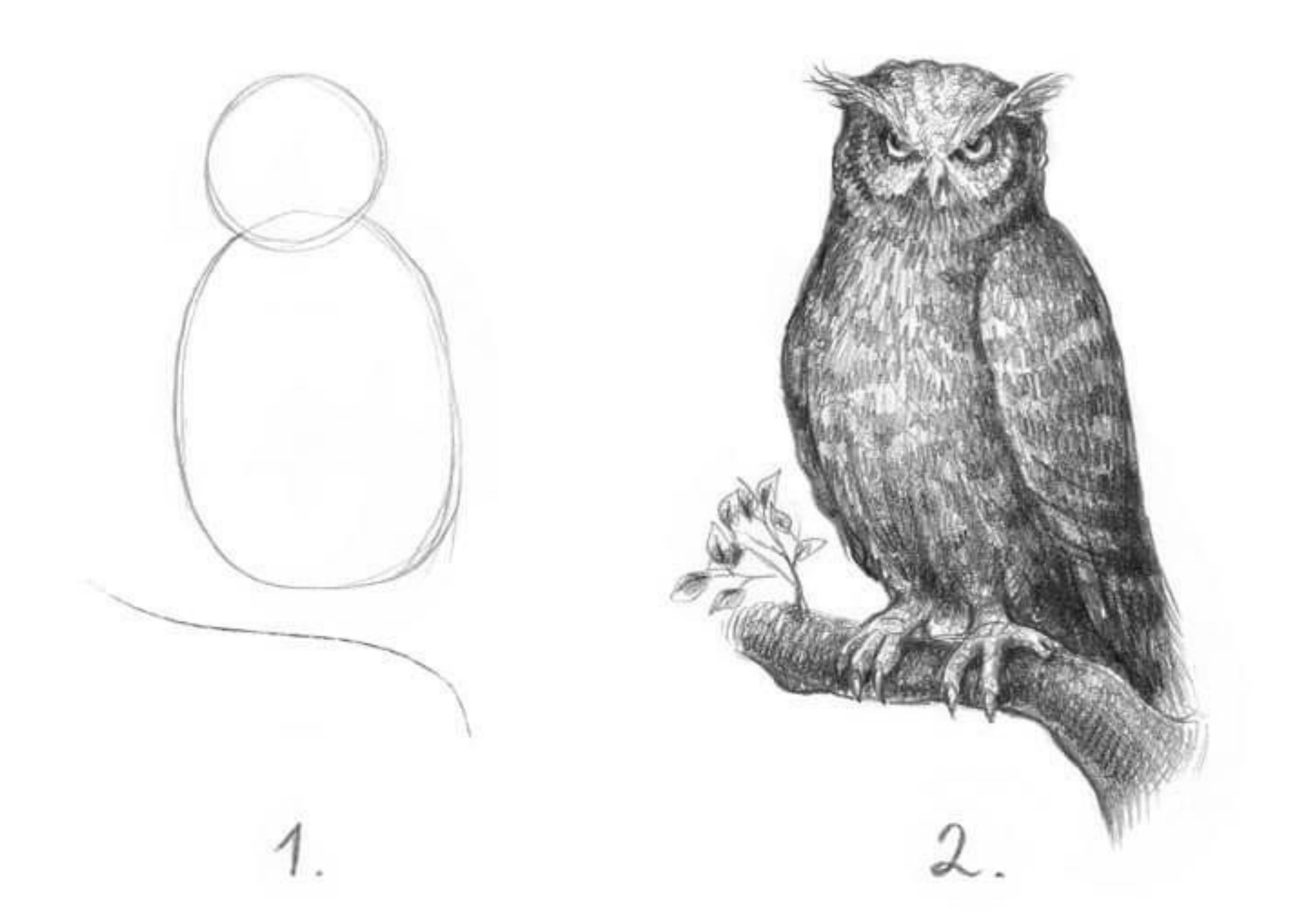
Users	Value
5015	3
5030	5

Indexation

OP	Users	Value
add	5015	1
add	5030	2
add	5015	2
sub	5030	2
add	5030	5

Indexation →

Users	Value
5015	3
5030	5

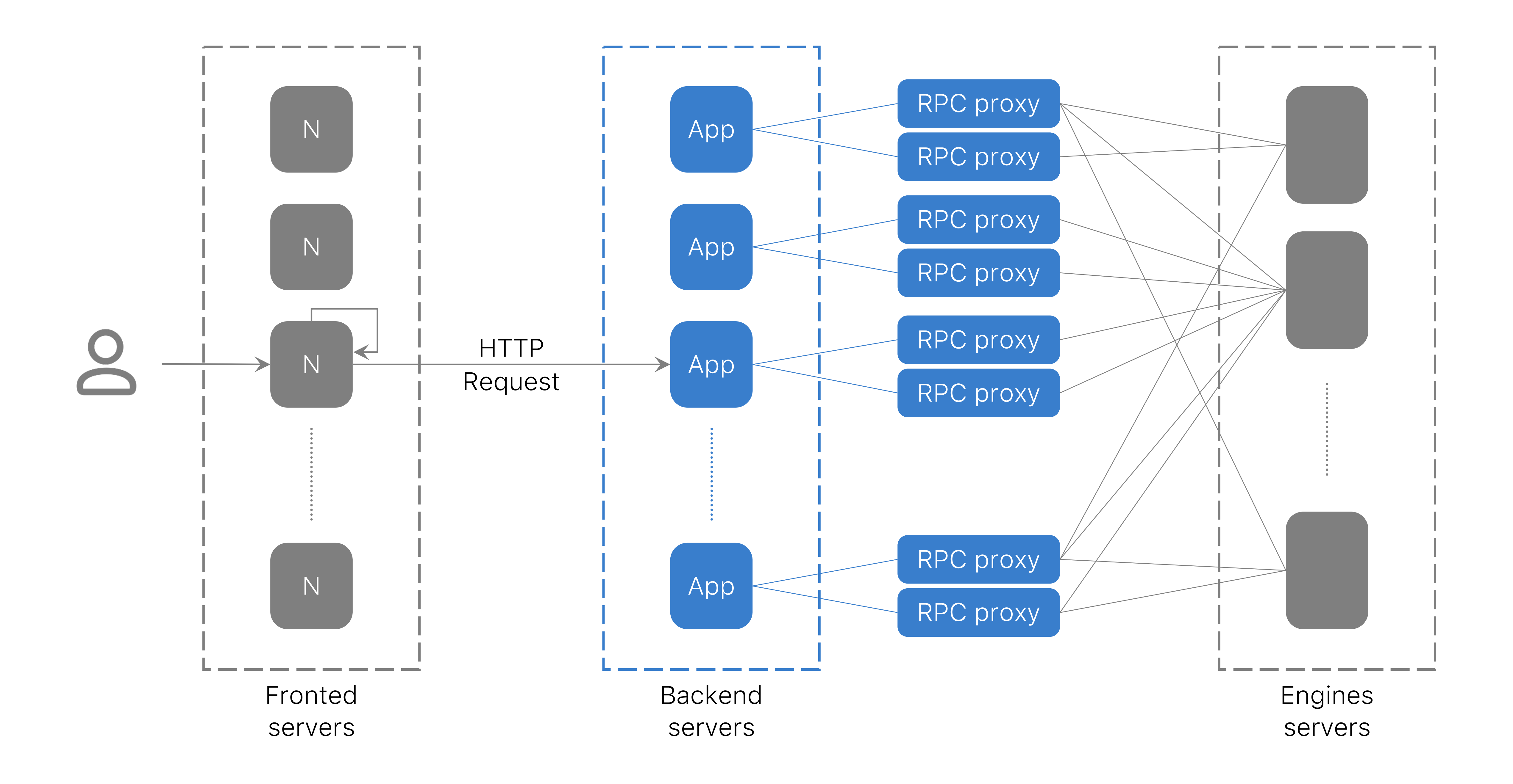

1.
2.

Особенности движков

1. Однопоточные*
2. Данные и бизнес-логика рядом
3. Чтений сильно больше записей
4. Индексируются в отдельном процессе, используя COW операционной системы
5. Данные персистентные, но в основном IN-MEMORY
6. Скорость важнее консистентности на чтение

RPC-proxy

основной компонент для роутинга
и контроля распространения запросов


N
N
N
N
HTTP
Request
App
App
App
App
RPC proxy
RPC proxy
RPC proxy
RPC proxy
RPC proxy
RPC proxy
RPC proxy
RPC proxy
Fronted
servers
Backend
servers
Engines
servers

Топология связей

1

PRC-proxy знает
про каждый кластер

2

PRC-proxy
знает про шарды

3

PRC-proxy установлен
локально к каждому
APP-серверу

4

Занимает один
хоп по сети



ПРОБЛЕМА mesh-топологии



При отказе одного из серверов общее время ответа кластера увеличивается



Начинает страдать время ответа API

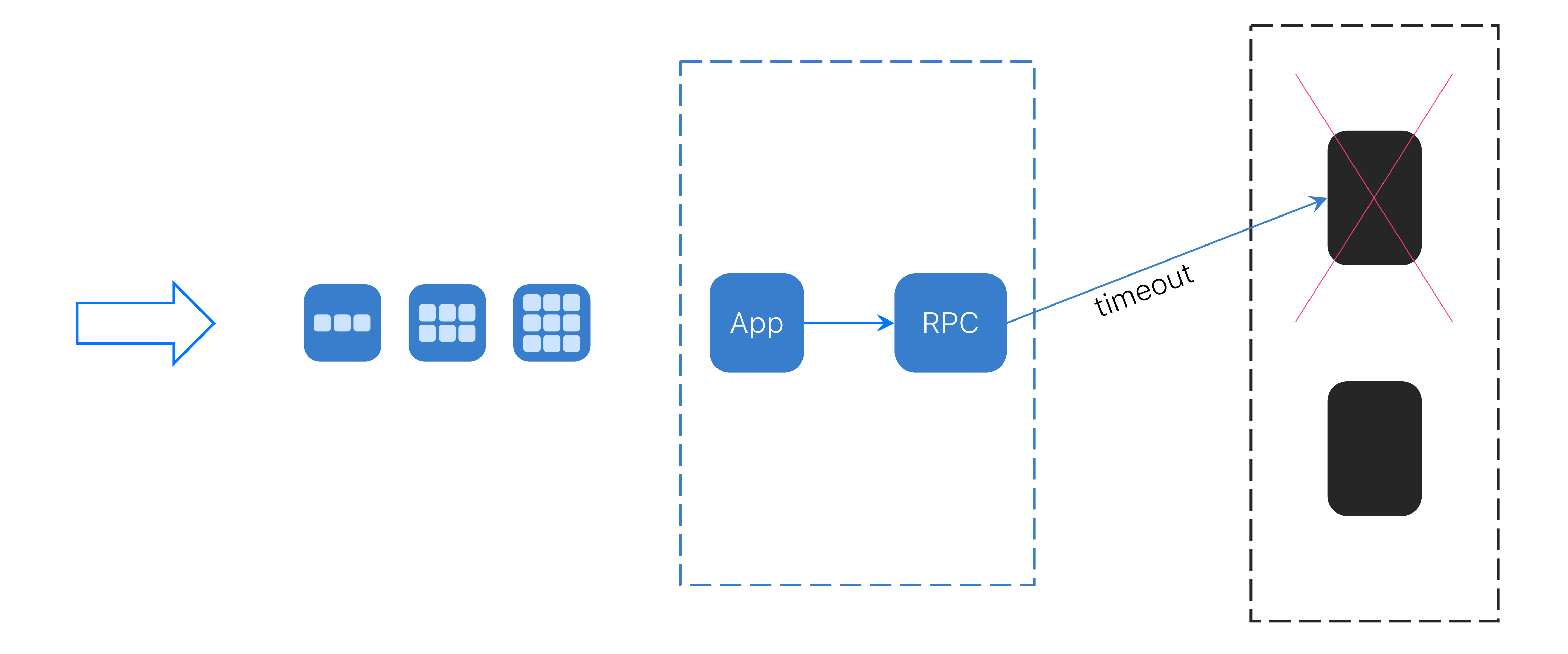
3

Заканчиваются ресурсы Backend-серверов

4

Получаем отказ в обслуживании всего сайта

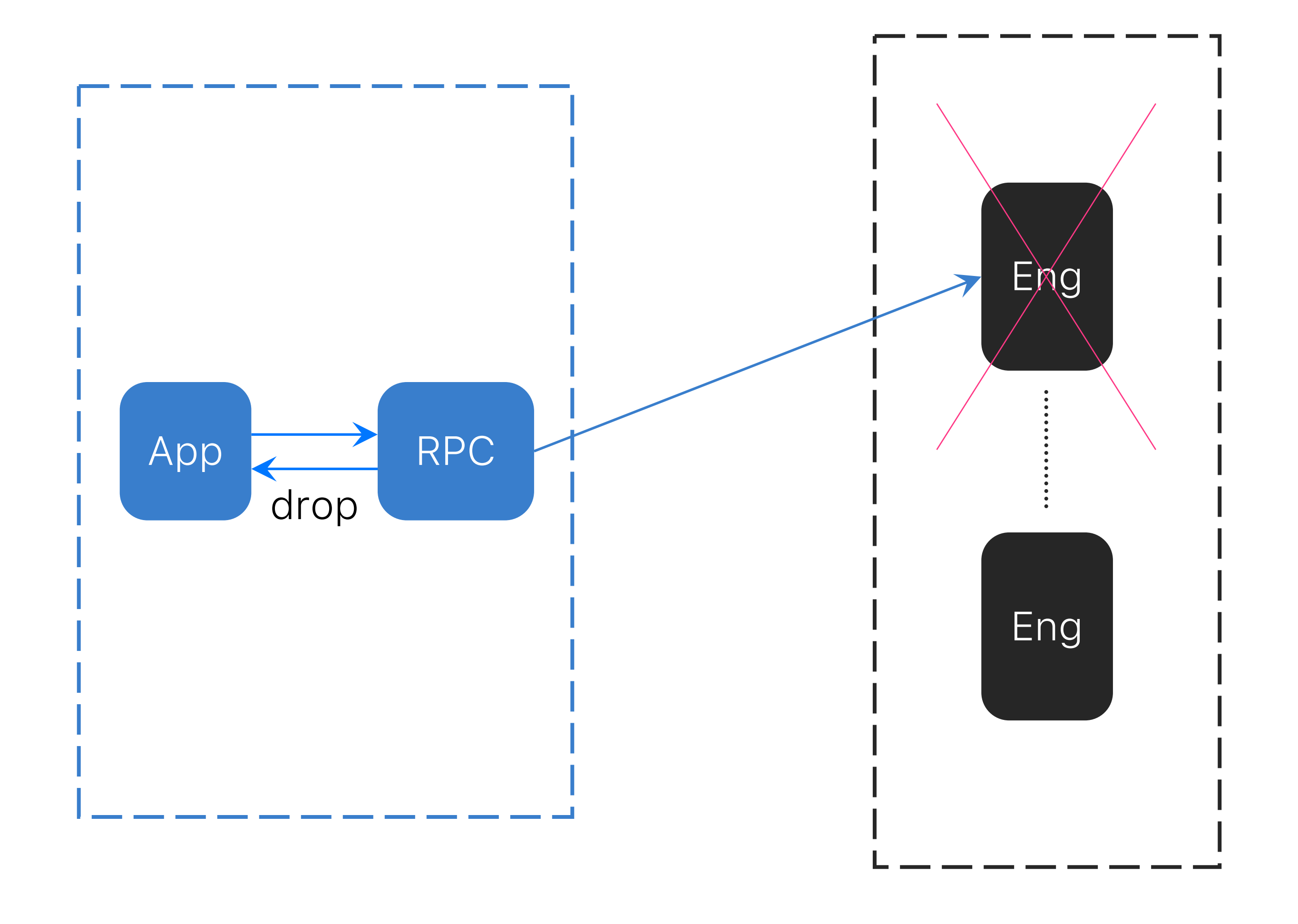


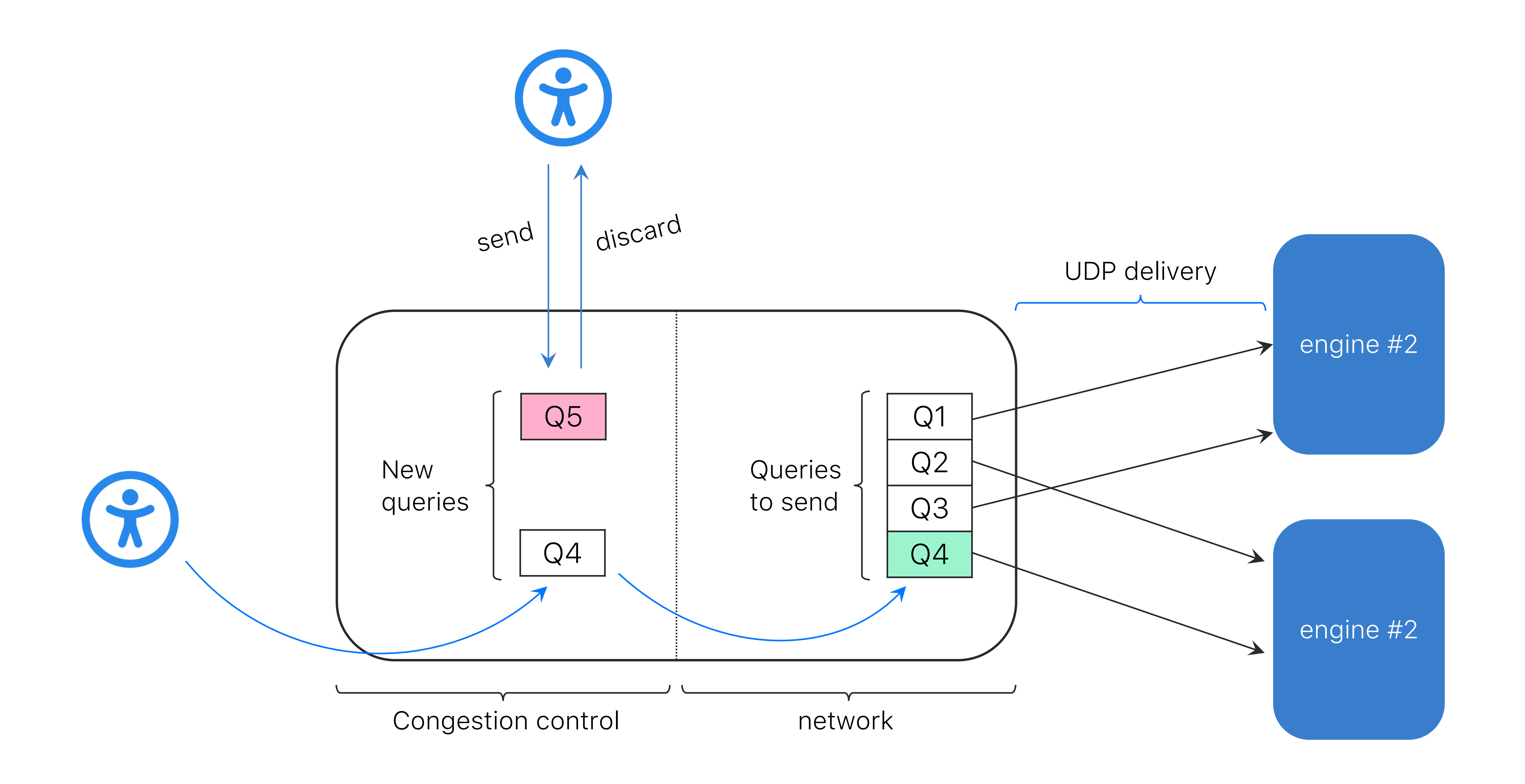

App
RPC
timeout

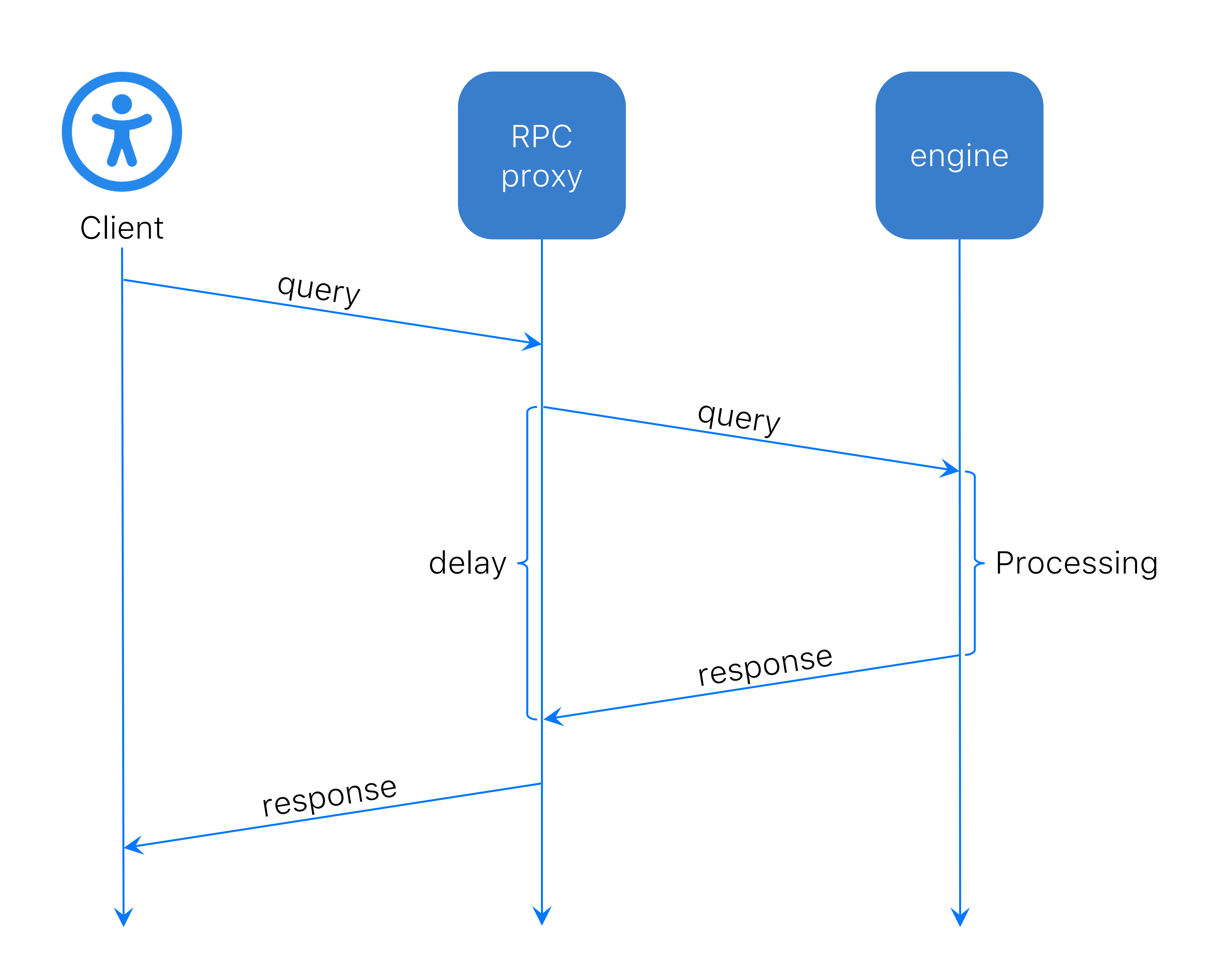
Решение — Congestion Control на application-уровне

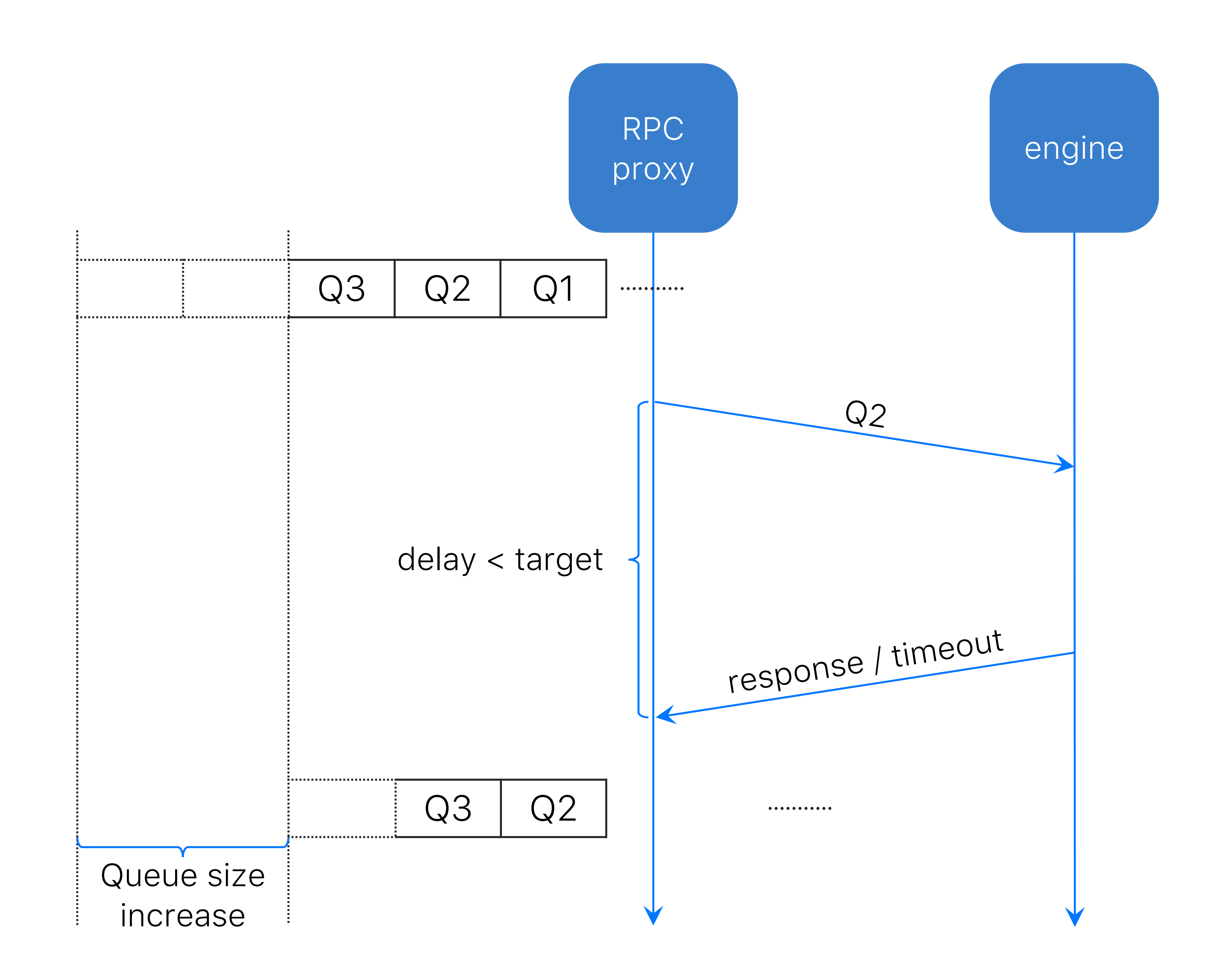
Идея: добавить на RPC-proxy CC на уровне запросов

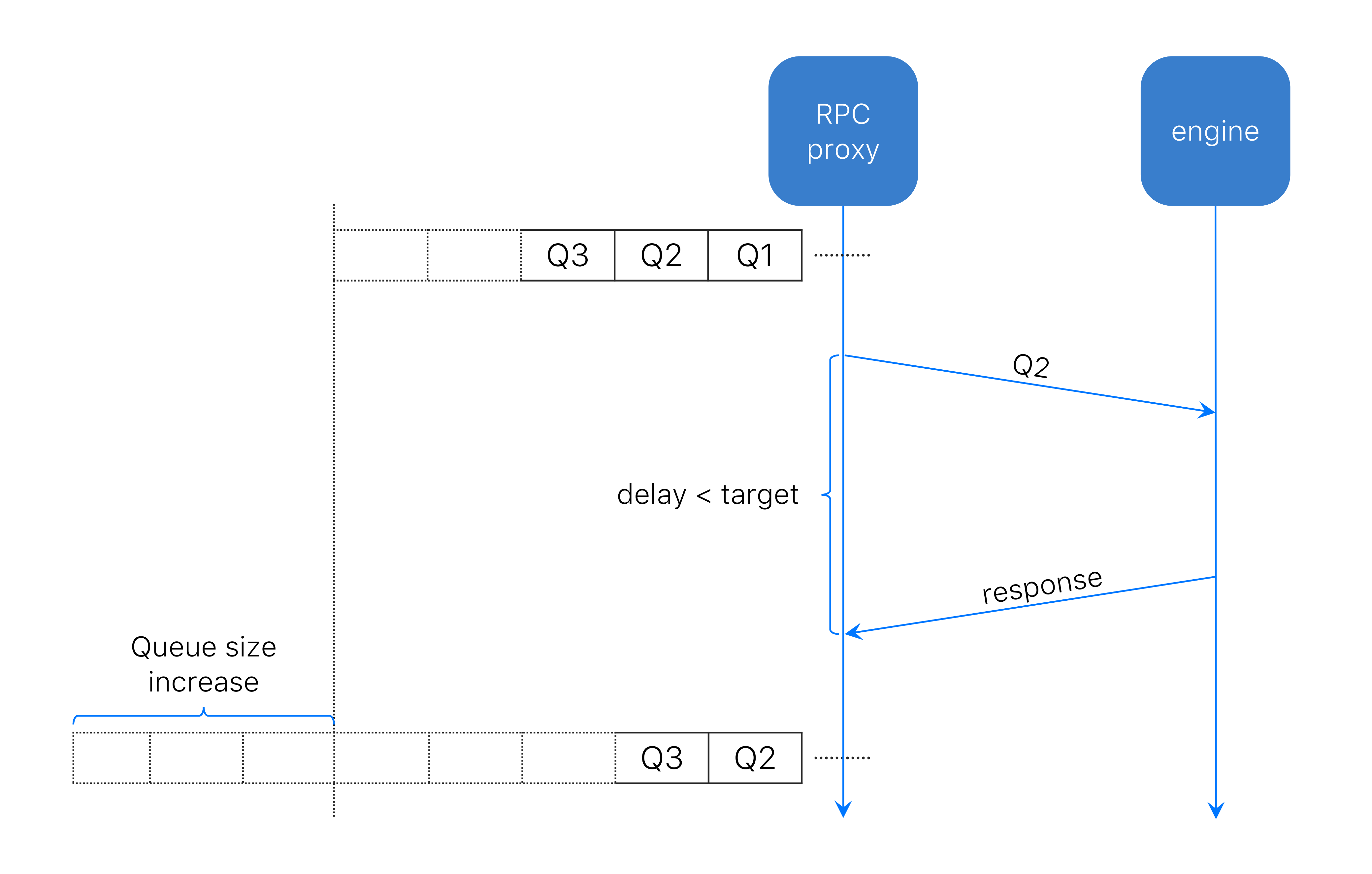
Если сервер начинает медленно отвечать — в него начинает приходить меньше запросов


App
RPC
drop
Eng
Eng

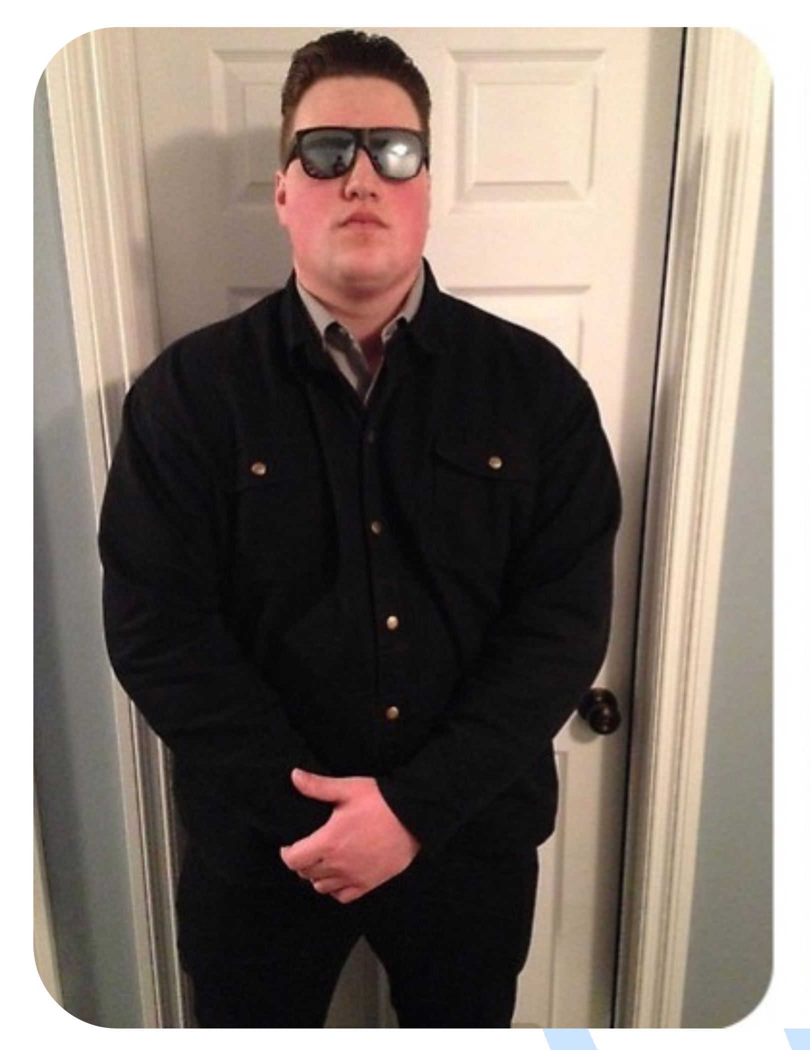

send
discard
UDP delivery
engine #2
Q5
New
queries
Q4
Queries
to send
Q1
Q2
Q3
Q4
engine #2
Congestion control
network


RPC
proxy
engine
Client
query
query
delay
Processing
response
response


RPC proxy
engine
Q3
Q2
Q1
Q2
delay < target
response / timeout
Q3
Q2
Queue size increase


RPC proxy
engine
Q3
Q2
Q1
Q2
delay < target
response
Queue size increase
Q3
Q2

discard

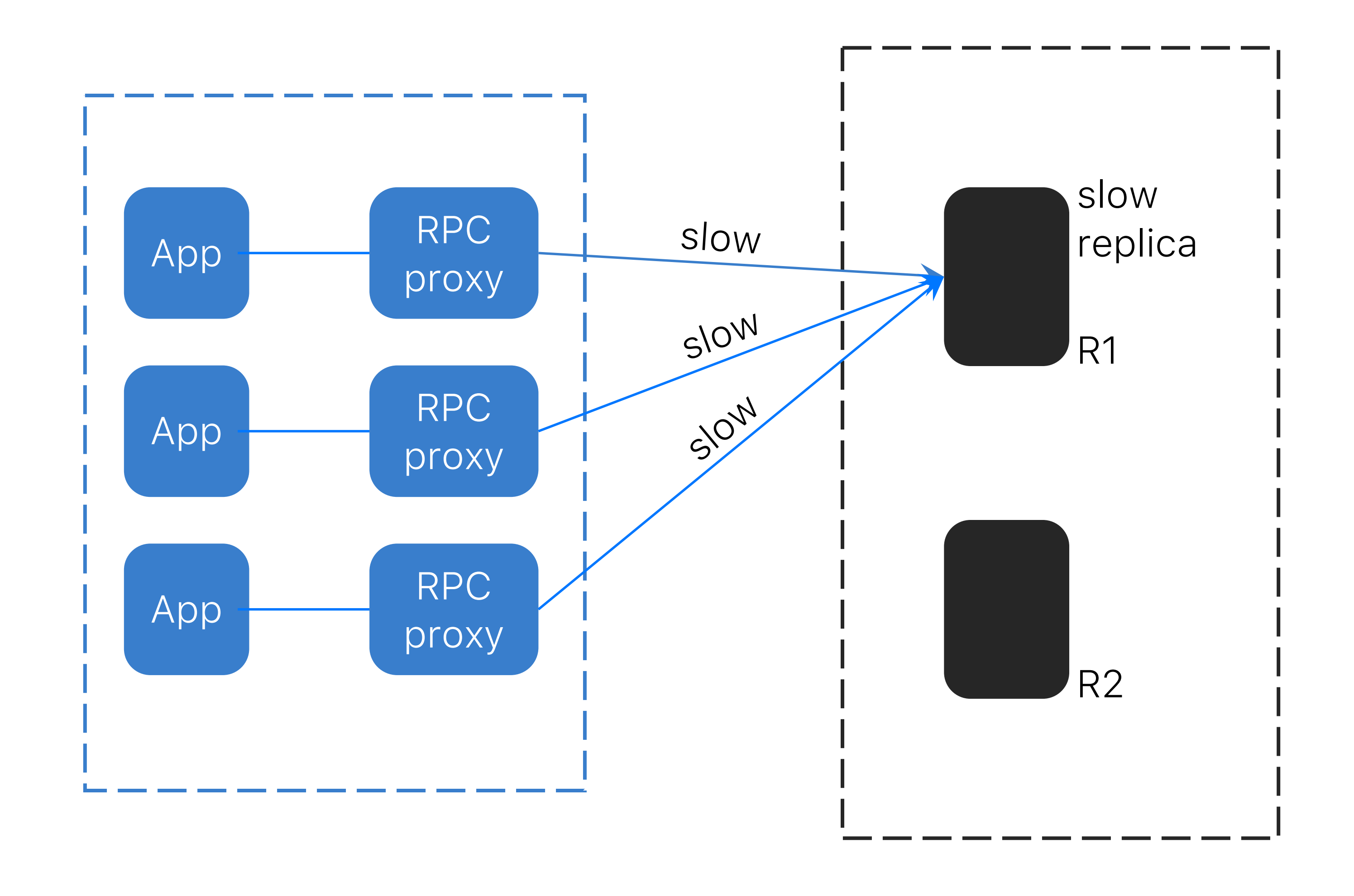


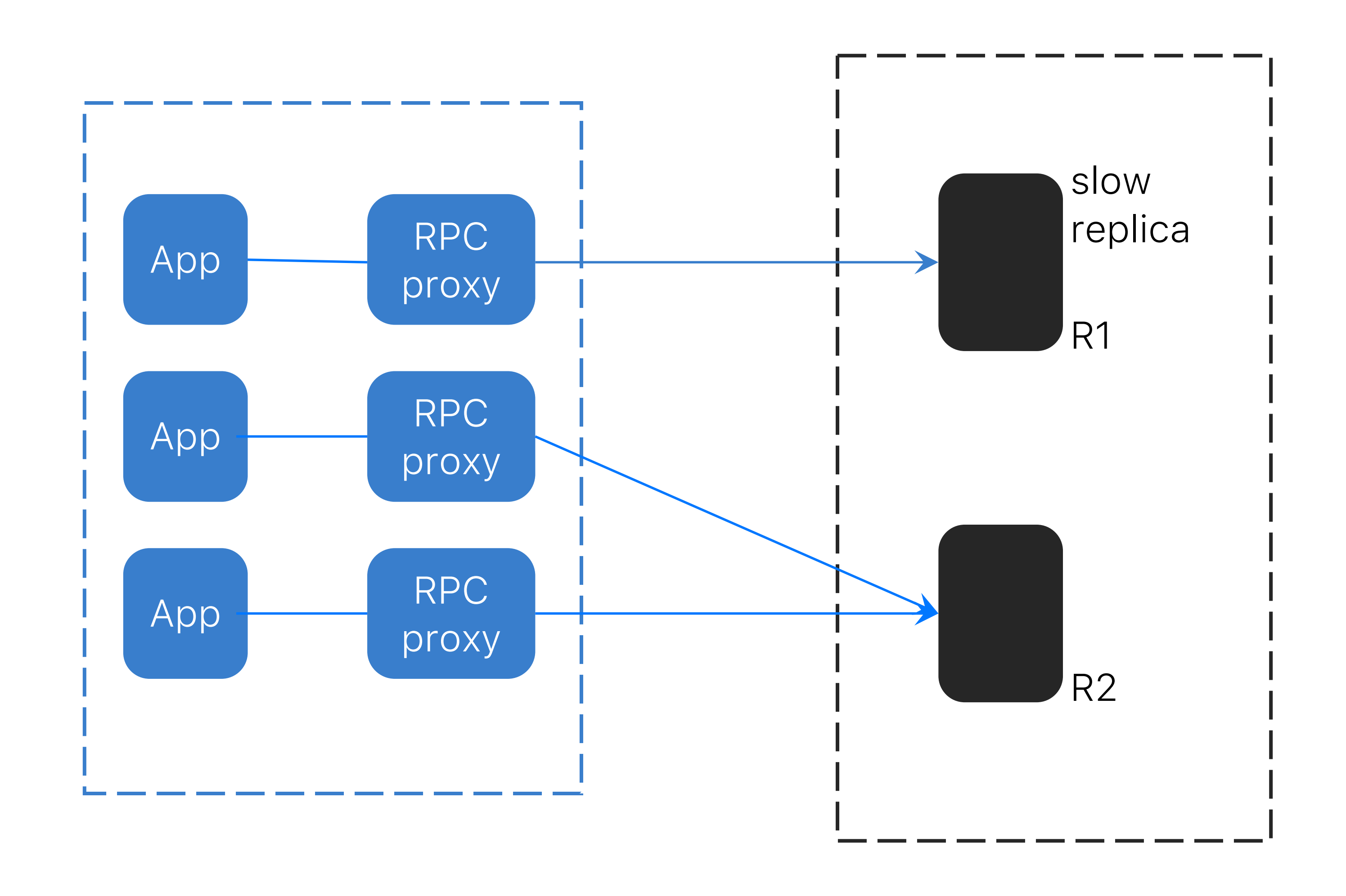
send



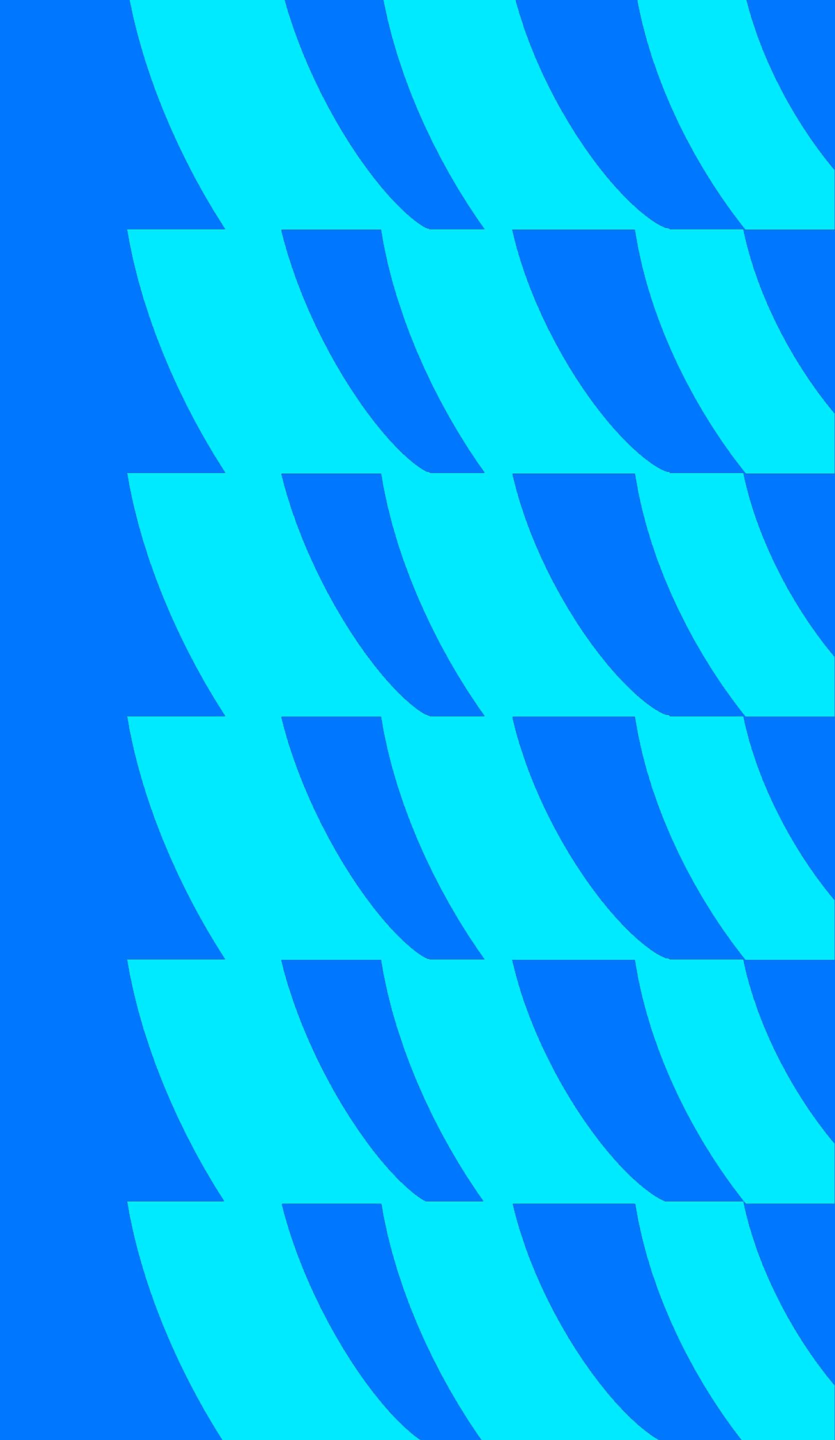
Application-level CC как балансер

Побочным эффектом является то, что механизмы CC позволяют уводить трафик с загруженной реплики


App
RPC proxy
App
RPC proxy
App
RPC proxy
slow
slow
slow
slow replica
R1
R2


App
RPC proxy
App
RPC proxy
App
RPC proxy
slow replica
R1
R2

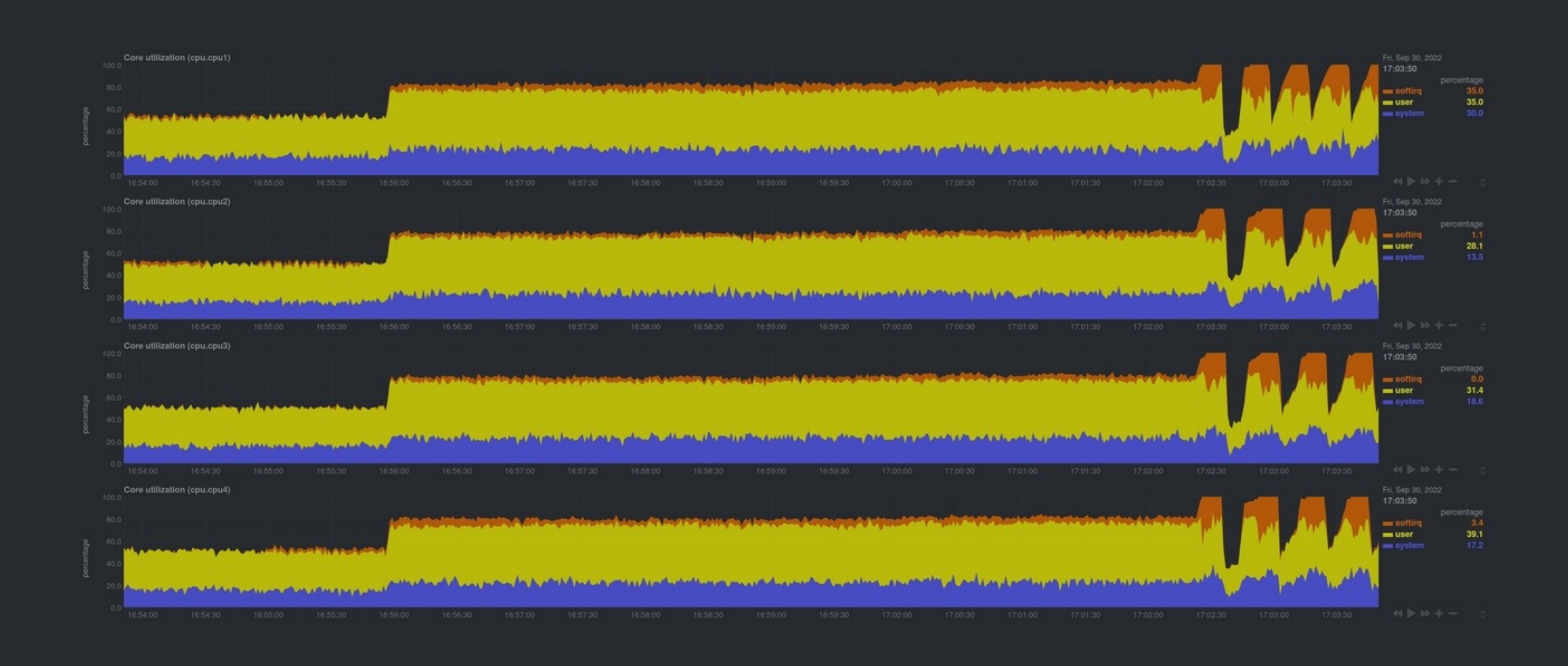
Результаты



При нагрузке в 100% CPU
и утилизации канала сервер
отбивал часть запросов,
при этом не повышая
задержку успешных

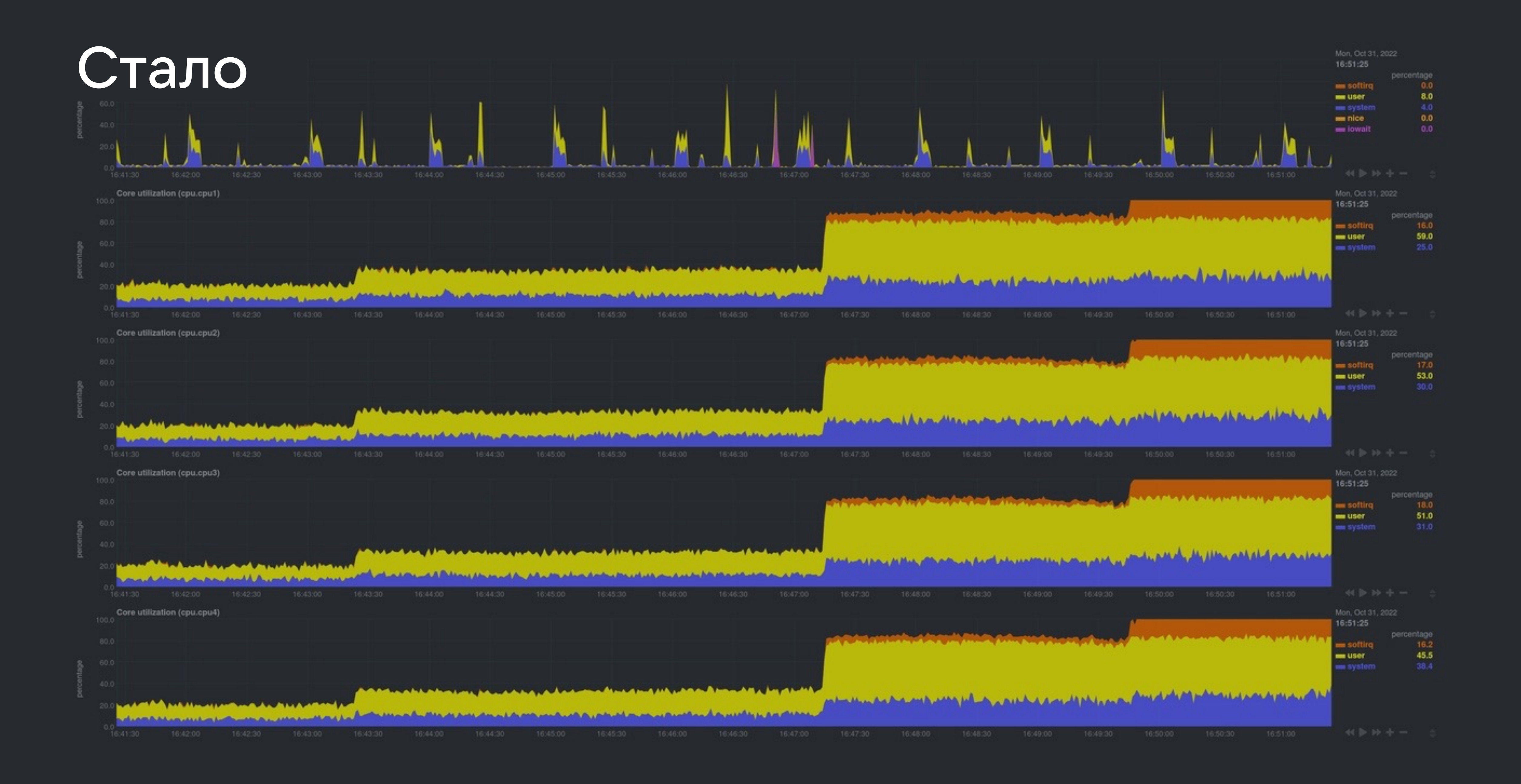
При замедлении реплики
часть запросов идет
на соседнюю, снижая
при этом нагрузку
на загруженную

Было

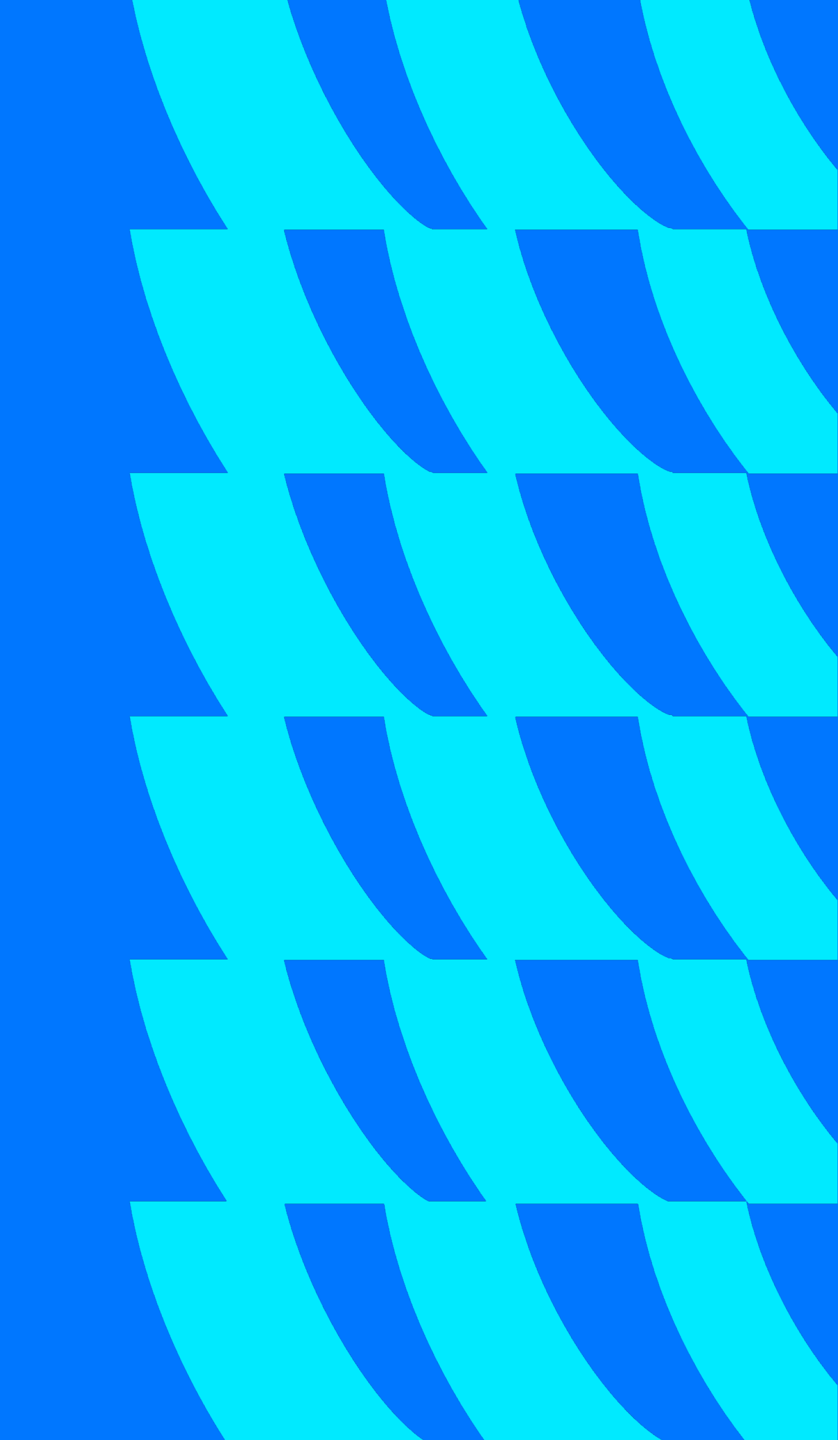


Core utilization (cpu.cpu1)
100.0
80.0
60.0
40.0
20.0
0.0
percentage
16:54:00 16:54:30 16:55:00 16:55:30 16:56:00 16:56:30 16:57:00 16:57:30 16:58:00 16:58:30 16:59:00 16:59:30 17:00:00 17:00:30 17:01:00 17:01:30 17:02:00 17:02:30 17:03:00 17:03:30
Fri, Sep 30, 2022
17:03:50
percentage
softirq 35.0
user 35.0
system 30.0
Core utilization (cpu.cpu2)
100.0
80.0
60.0
40.0
20.0
0.0
percentage
16:54:00 16:54:30 16:55:00 16:55:30 16:56:00 16:56:30 16:57:00 16:57:30 16:58:00 16:58:30 16:59:00 16:59:30 17:00:00 17:00:30 17:01:00 17:01:30 17:02:00 17:02:30 17:03:00 17:03:30
Fri, Sep 30, 2022
17:03:50
percentage
softirq 1.1
user 28.1
system 13.5
Core utilization (cpu.cpu3)
100.0
80.0
60.0
40.0
20.0
0.0
percentage
16:54:00 16:54:30 16:55:00 16:55:30 16:56:00 16:56:30 16:57:00 16:57:30 16:58:00 16:58:30 16:59:00 16:59:30 17:00:00 17:00:30 17:01:00 17:01:30 17:02:00 17:02:30 17:03:00 17:03:30
Fri, Sep 30, 2022
17:03:50
percentage
softirq 0.0
user 31.4
system 18.6
Core utilization (cpu.cpu4)
100.0
80.0
60.0
40.0
20.0
0.0
percentage
16:54:00 16:54:30 16:55:00 16:55:30 16:56:00 16:56:30 16:57:00 16:57:30 16:58:00 16:58:30 16:59:00 16:59:30 17:00:00 17:00:30 17:01:00 17:01:30 17:02:00 17:02:30 17:03:00 17:03:30
Fri, Sep 30, 2022
17:03:50
percentage
softirq 3.4
user 39.1
system 17.2

Стало


Mon, Oct 31, 2022
16:51:25
percentage
softirq 0.0
user 8.0
system 4.0
nice 0.0
iowait 0.0
Core utilization (cpu.cpu1)
Mon, Oct 31, 2022
16:51:25
percentage
softirq 16.0
user 59.0
system 25.0
Core utilization (cpu.cpu2)
Mon, Oct 31, 2022
16:51:25
percentage
softirq 17.0
user 53.0
system 30.0
Core utilization (cpu.cpu3)
Mon, Oct 31, 2022
16:51:25
percentage
softirq 18.0
user 51.0
system 31.0
Core utilization (cpu.cpu4)
Mon, Oct 31, 2022
16:51:25
percentage
softirq 16.2
user 45.5
system 38.4

Некоторые трюки с мемкешом





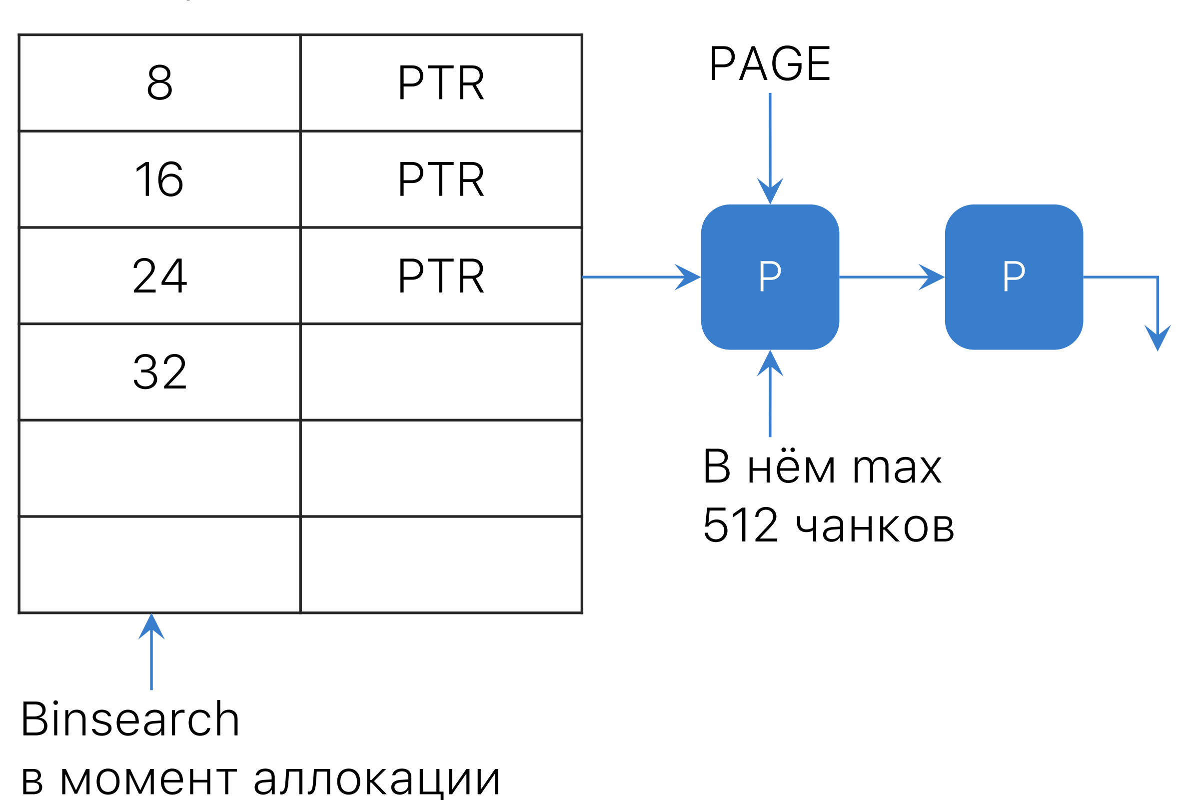
Memcached-shm

Порой OOM-киллер или какая-то другая причина забирает лучших

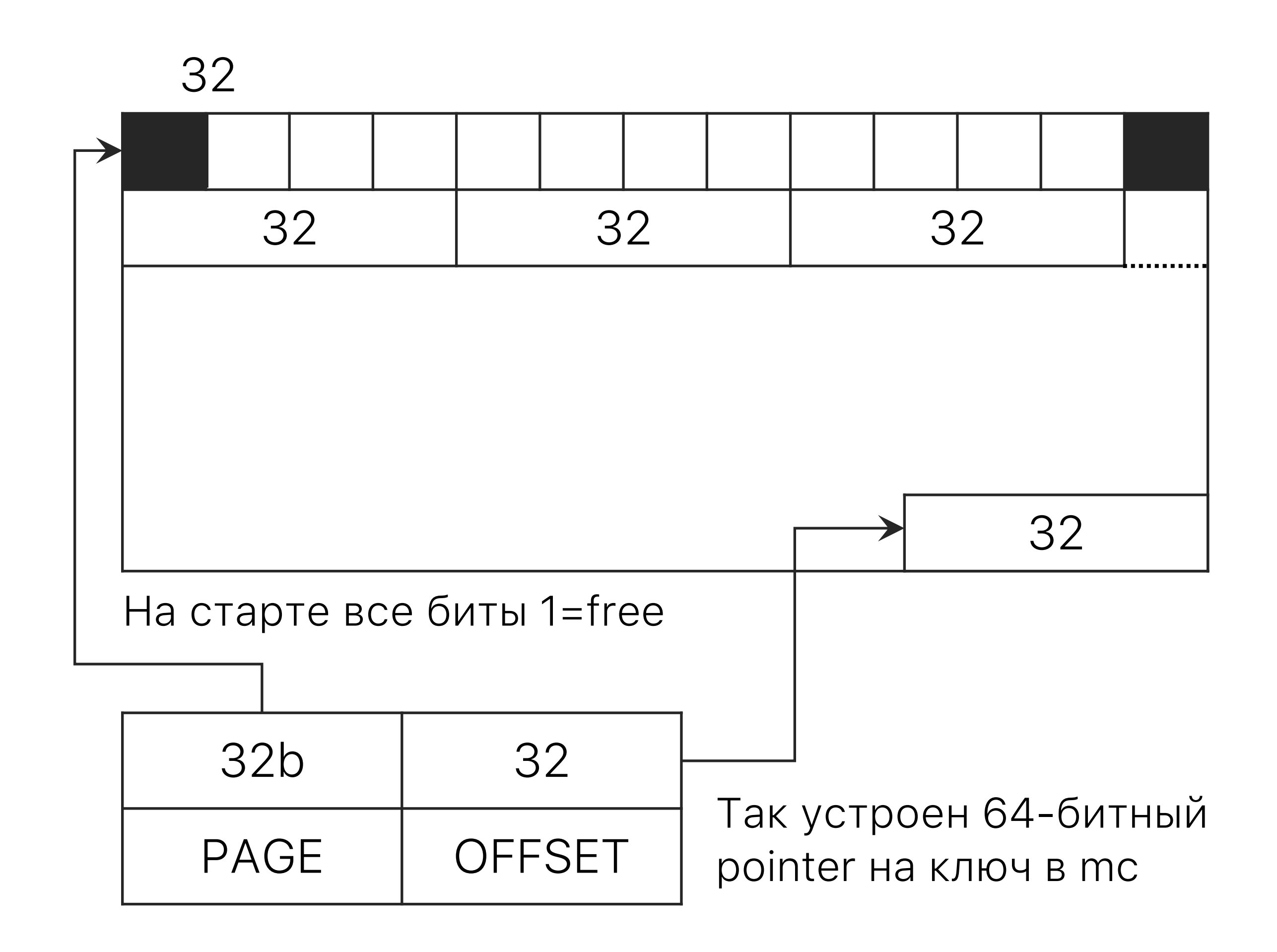
Для того чтобы не было так, был реализован простой аллокатор, использующий shared-memory



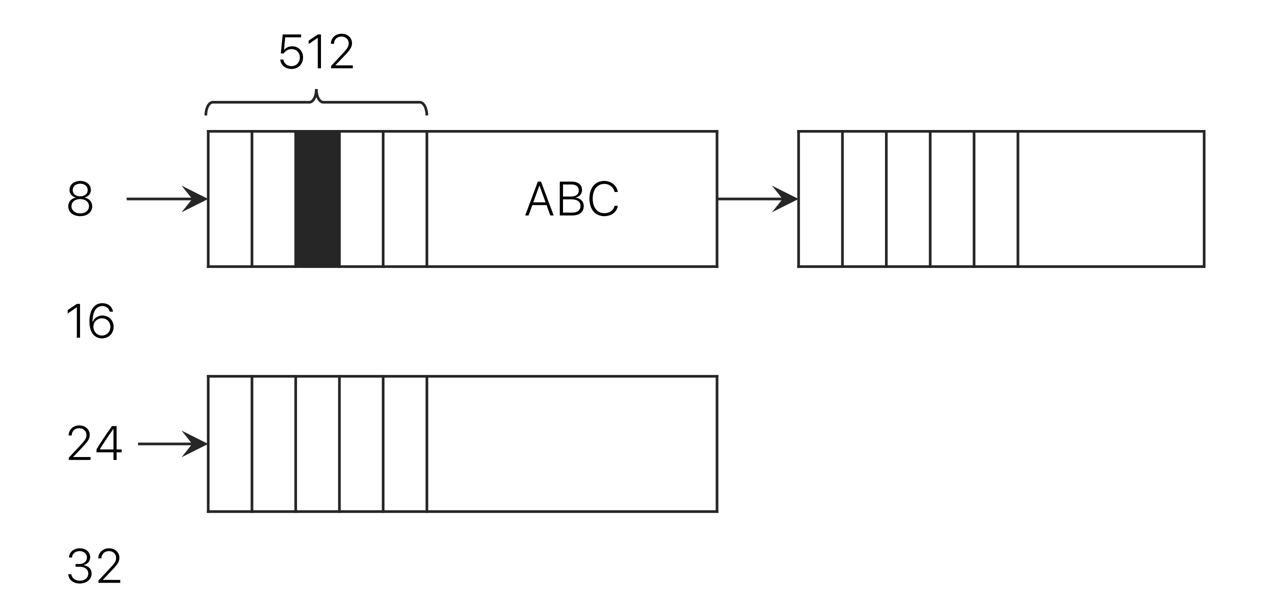
Memcached-shm: allocator



Memcached-shm: allocator



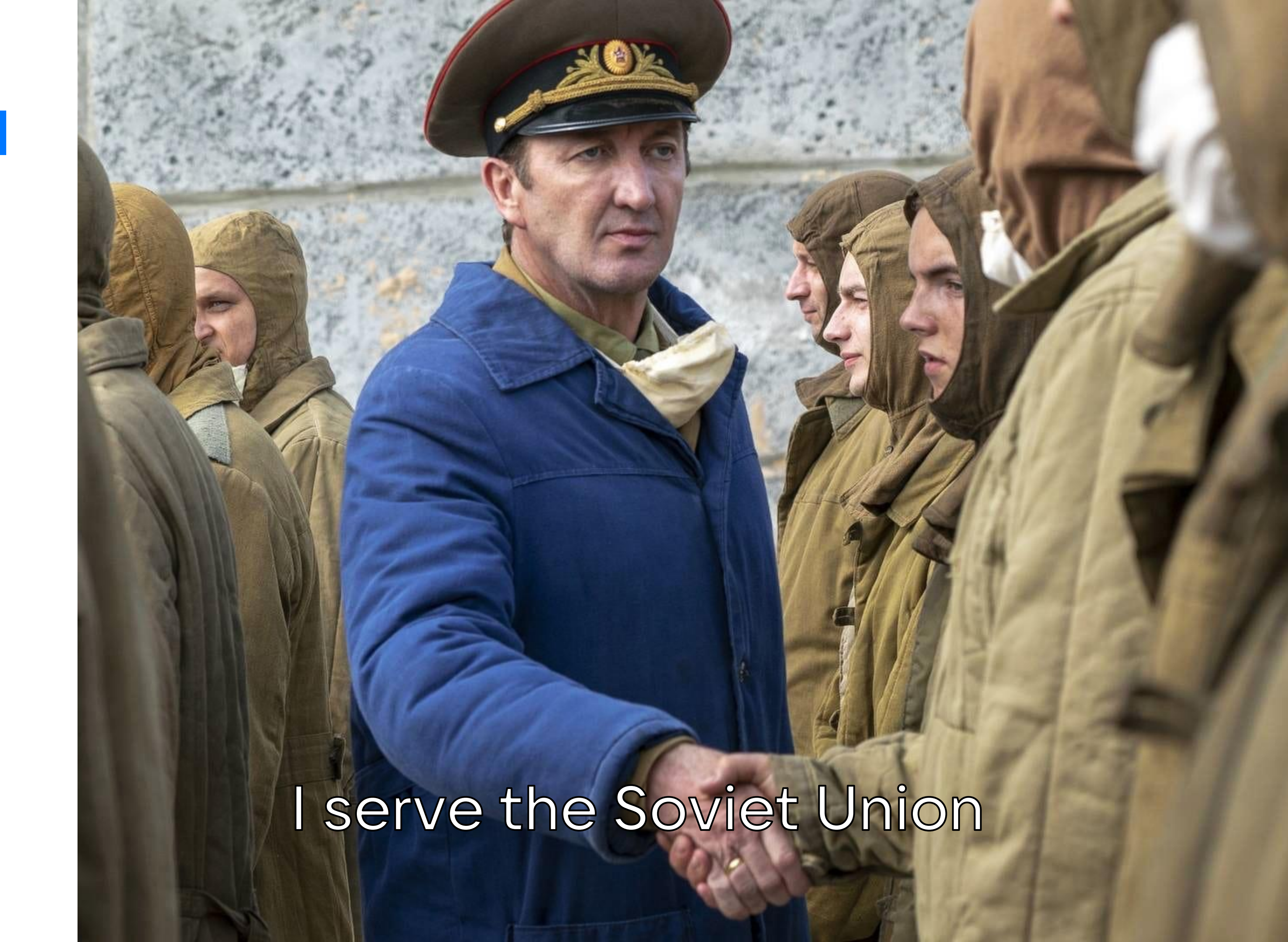
Memcached-shm: allocator



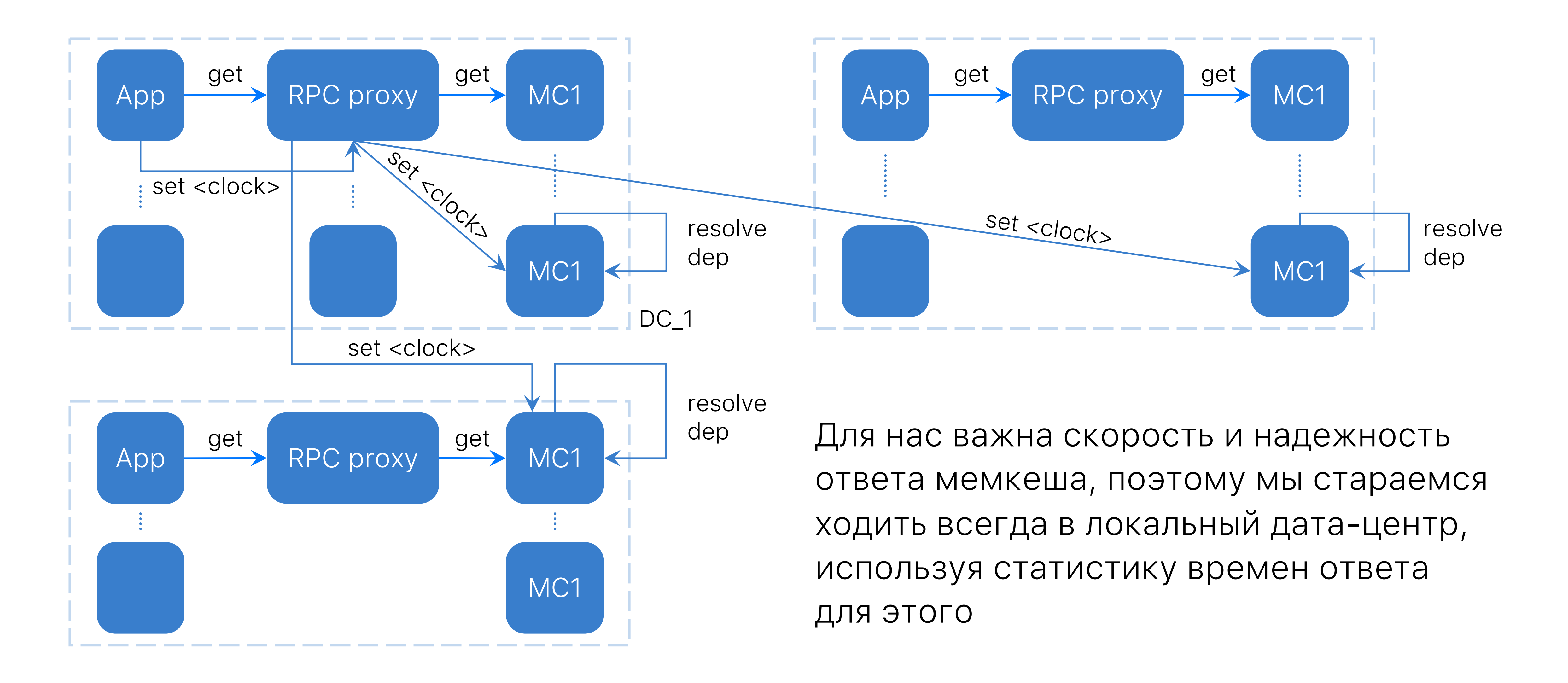
Memcached-shm: статистика аллокаций

sizeclass	lru_size_evaluation	lru_updates	lru_drops	lru_drops_ffff
0	0	0	0	0
1	6107	504133	0	0
2	82140	13986554	11698	11698
3	1788281	94169784	483222	483222
4	2849519	53064428	1551886	748672
5	643132	24328205	871995	163772
6	249453	23920864	284862	58490
7	456909	12624350	340068	116980
8	343046	28121403	567001	81886
9	53230	4440532	56235	11698
10	74278	4987243	20728	11698
11	88363	11146533	43233	17663
12	48847	10969569	35208	11698
13	88861	11663046	36099	23384
14	44527	7182312	12624	11698
15	59629	17415728	12251	11698
16	38810	6643563	21207	0

How I feel when I share anything

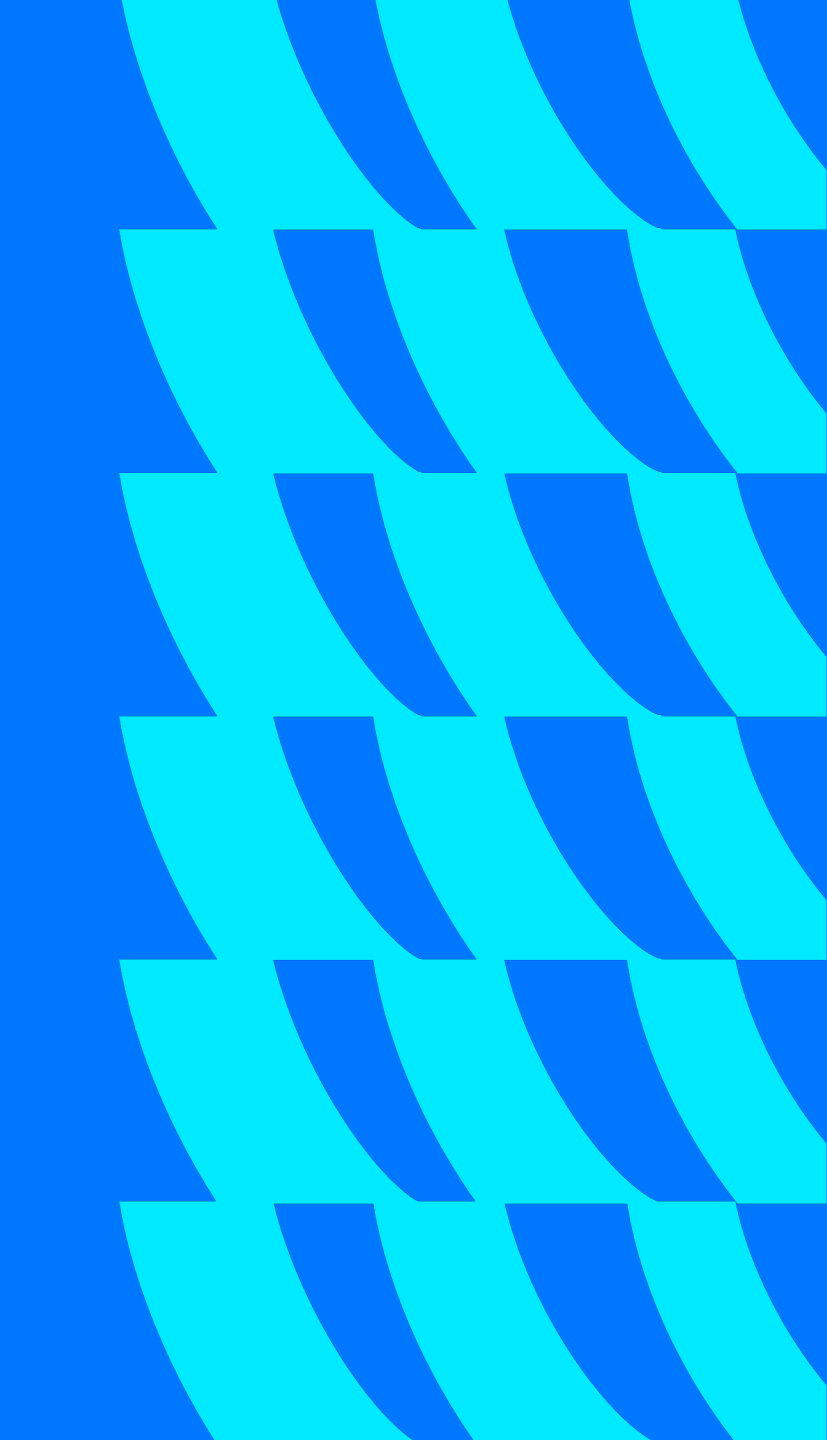


Memcached-replication

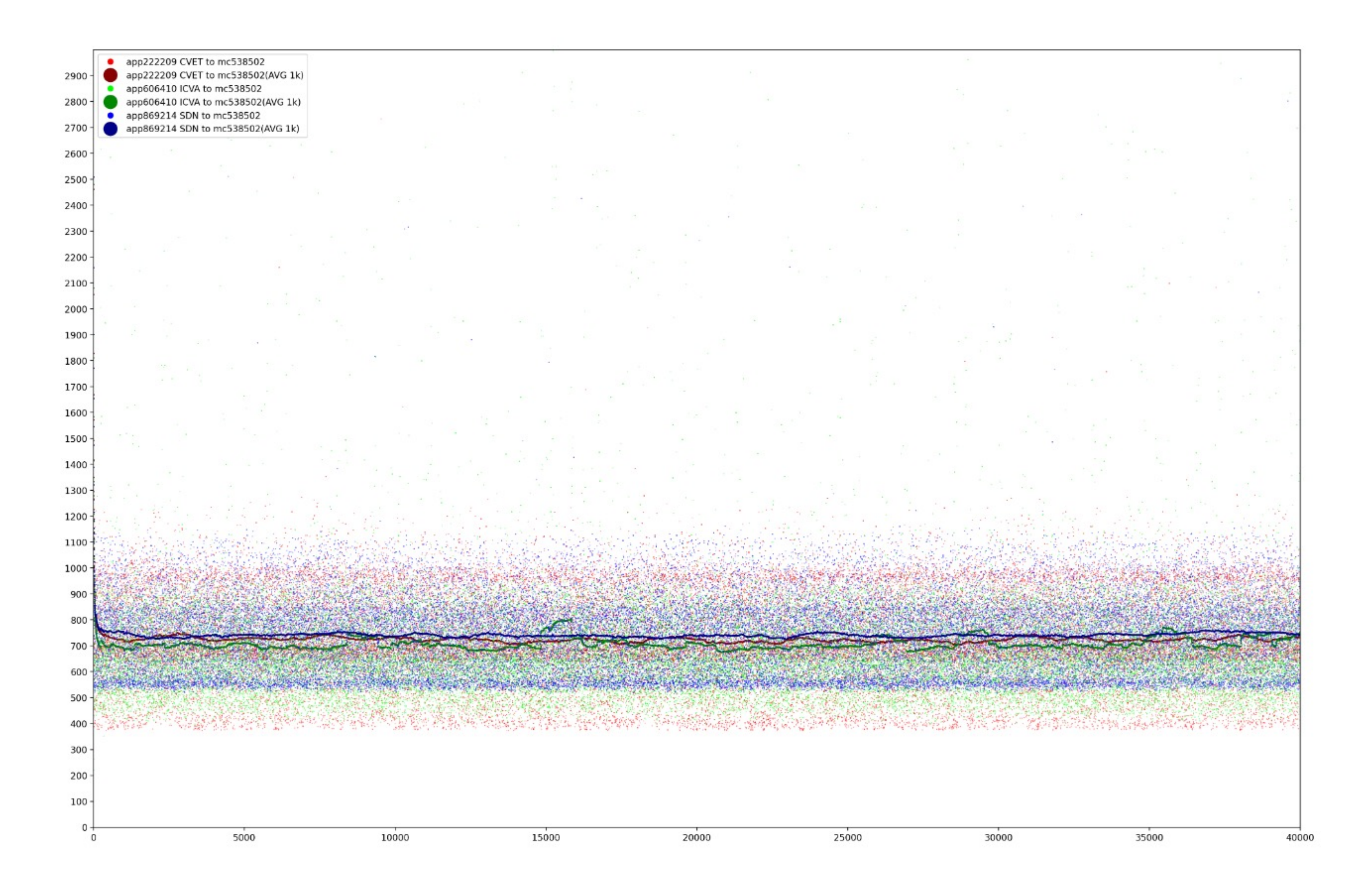


Для нас важна скорость и надежность ответа мемкеша, поэтому мы стараемся ходить всегда в локальный дата-центр, используя статистику времен ответа для этого

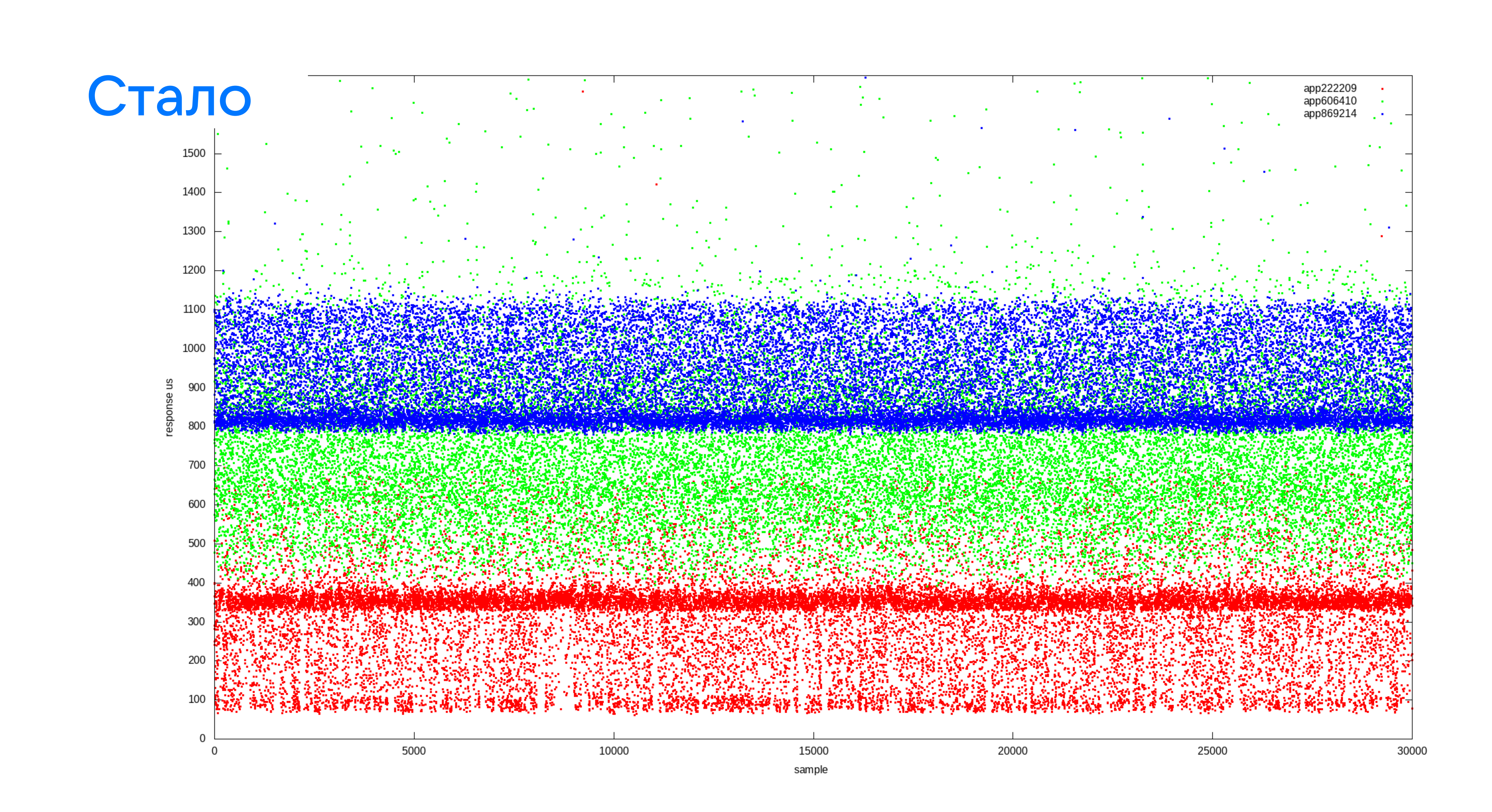
Почему mc-replicated – это важно?



Было

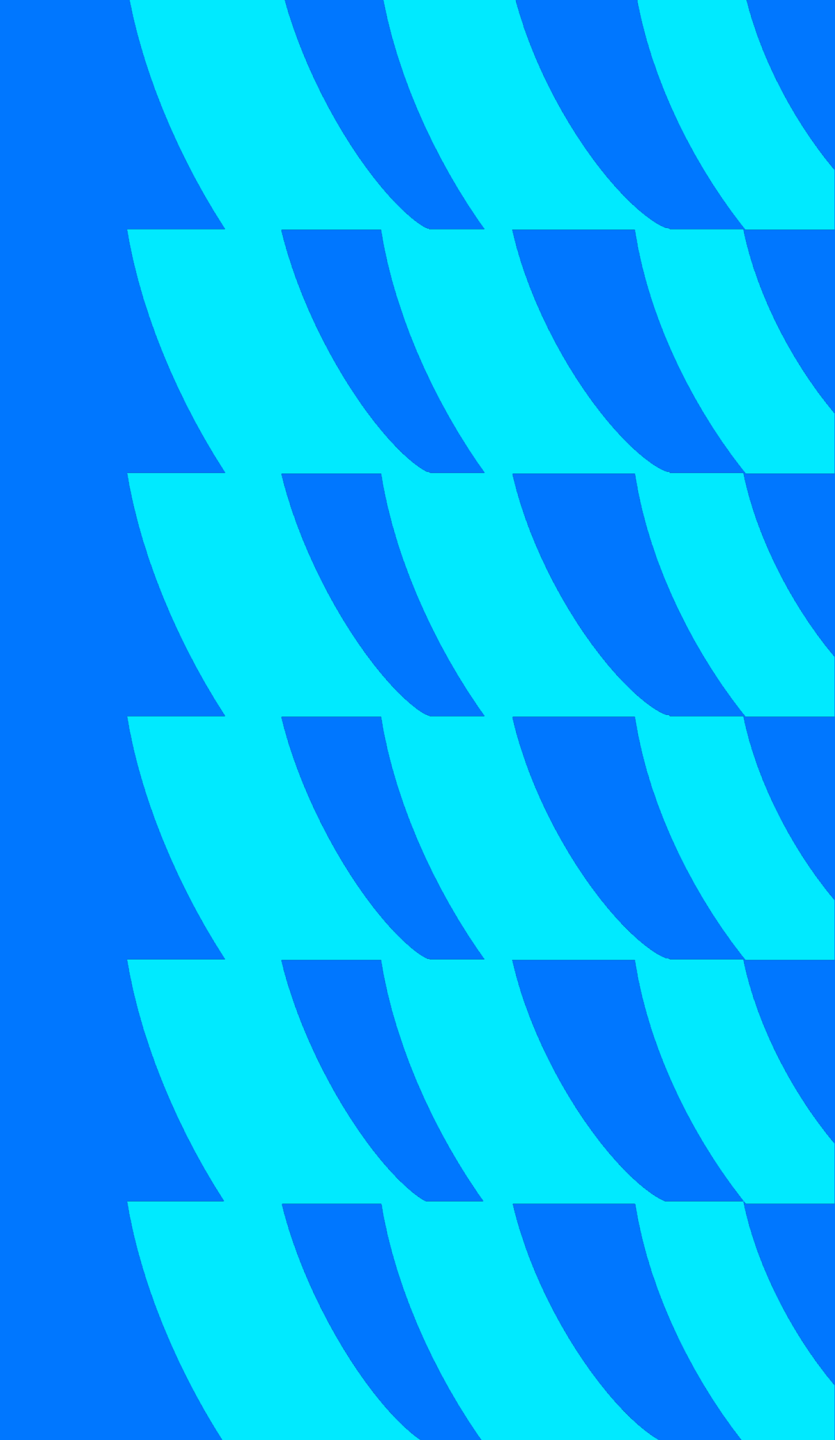

app222209 CVET to mc538502
app222209 CVET to mc538502(AVG 1k)
app606410 ICVA to mc538502
app606410 ICVA to mc538502(AVG 1k)
app869214 SDN to mc538502
app869214 SDN to mc538502(AVG 1k)

Стало



app222209
app606410
app869214
response us
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
sample
0
5000
10000
15000
20000
25000
30000

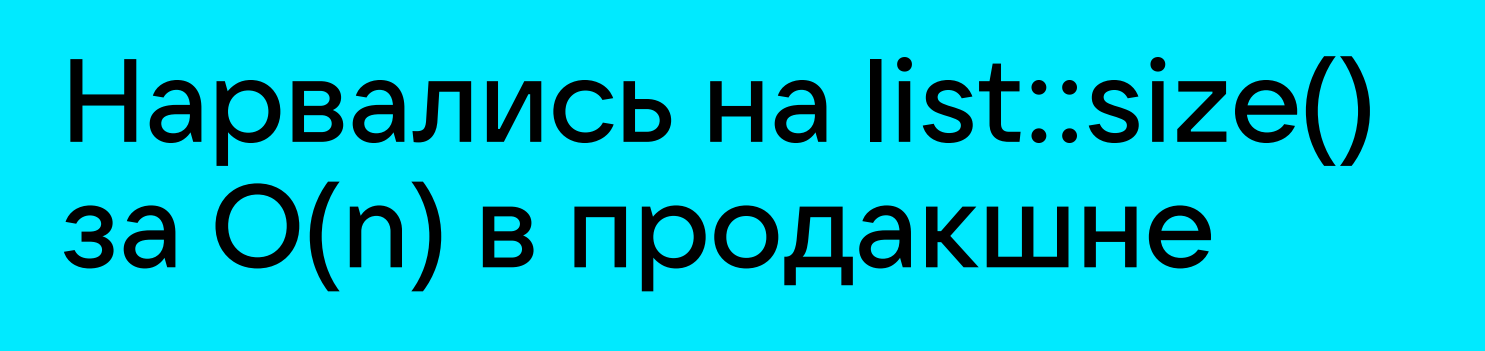
Бывало разное




РВАНЁТ
НЕ ДОЛЖНО...

When you watch the video at 2x speed to save time but it is still O(n)




Нарвались на list::size()
за O(n) в продакшне



Проблема

Часть кода собиралась старым компилятором, где несмотря на 11-ый стандарт, функция взятия размера выполнялась за линию

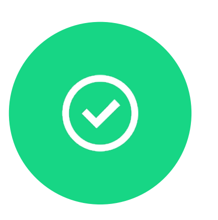
Что не преминуло сказаться на графиках



Проблема

Часть кода собиралась старым компилятором, где несмотря на 11-ый стандарт, функция взятия размера выполнялась за линию

Что не преминуло сказаться на графиках



Решение

Сейчас перешли на 20-ый стандарт и новые компиляторы, а для старых сборок где нужно подкладываем сборную солянку из старого glibc

Да, контейнеров у нас для движков нет

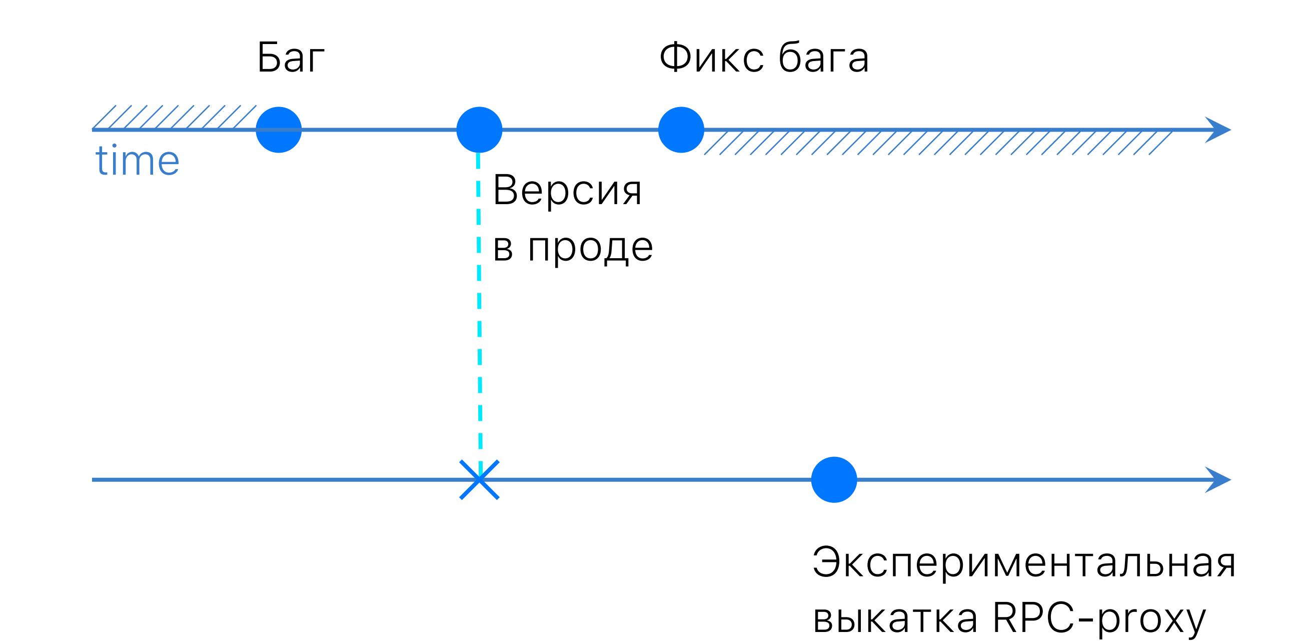


ха, небольшая задача на проксе
RPC-PROXY
пару дней и в прод
БАГ В MEMECACHED
ПАКЕТ СМЕРТИ ОТ RPC-PROXY

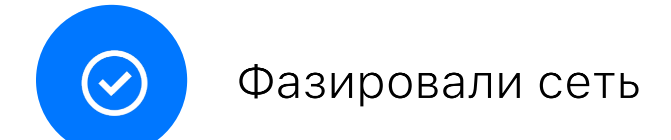
«Пакет смерти»

Самолет не падает при отказе одного двигателя

Так и здесь – при определенной конфигурации и определенной версии нашелся пакет, который вынес нам кеширующий слой

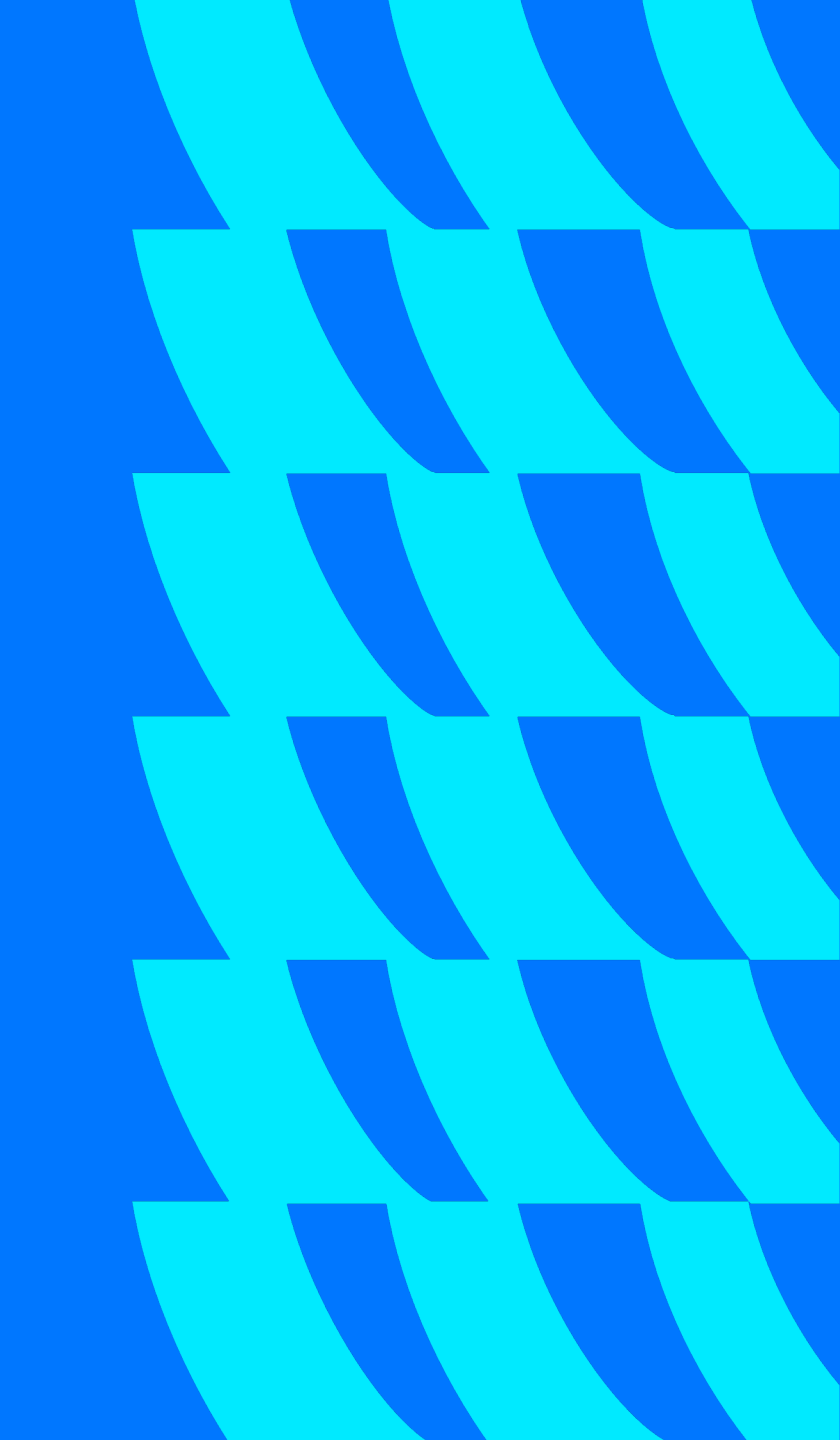


«Пакет смерти»: решение



- Фазировали сеть
- Обновили везде критичные компоненты
- Сделали устойчивый к падениям кеширующий слой

Хозяйке на заметку



Про компактность объекта

Не забывай про выравнивание
и сортировку типов в объекте

Кеш скажет спасибо

AVX-инструкции часто могут дать выигрыш в несколько раз

Argon2 крайне чувствителен к использованию векторных инструкций — использование AVX-512F ускоряет его работу в три раза по отношению к обычной реализации

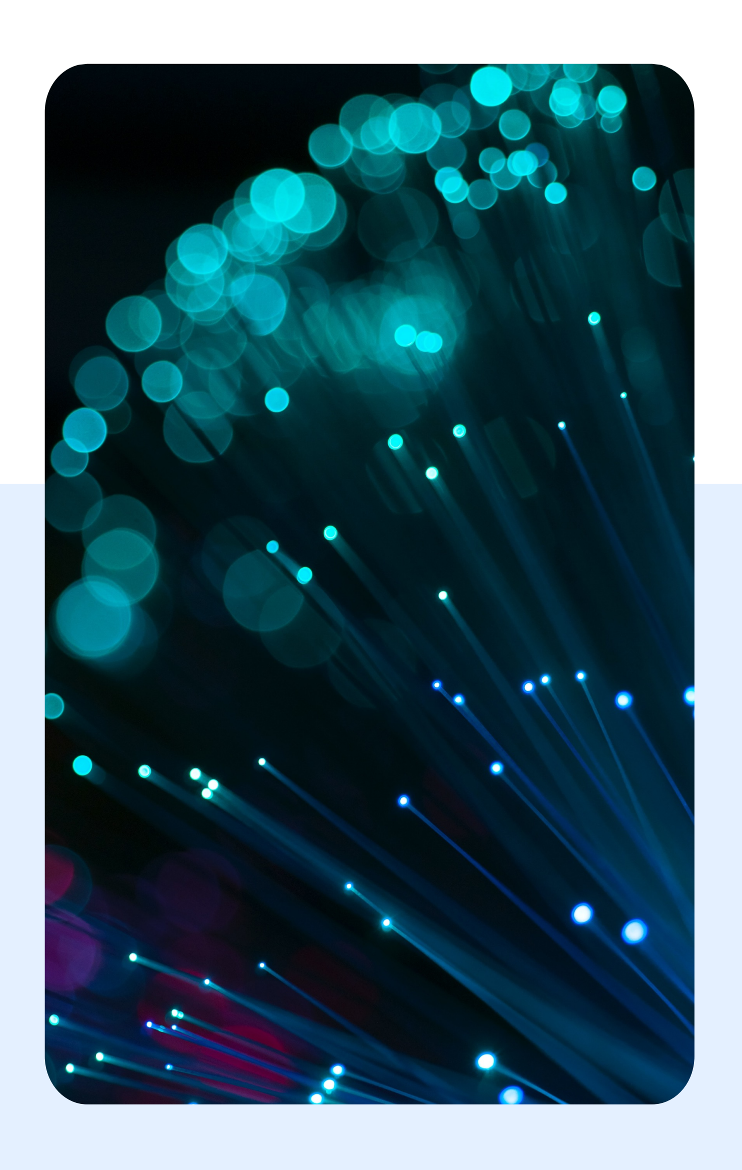
Переходы по указателям НЕ БЕСПЛАТНЫ

1. Итерация в b-дереве в 10x быстрее чем в set
2. Локальность обращений к памяти очень важна
3. Заменили стандартный std::set на b-дерево

b-дерево вместо std::set

Вся служебная информация в вершине b-дерева занимает 8 байт

- Размер поддерева вершины MAX 2^31 [0:0, 3:0]
- Капасити вершины [3:1, 4,6]
- Фактический размер вершины [4:7, 6:2]
- Бит листа [6:3]



b-дерево вместо std::set || Профит



При факторе B=32 на один инстанс мы тратим 4,8 байт, а сет, например, 40 байт

Используем устройство указателей для тегирования

Причем используем это двунаправленно, младшие три бита указателя из-за смещения также будут всегда нули

Старшие шестнадцать для решения ABA-проблемы



Использование UDP-based-протокола в DC позволяет снизить задержку и сэкономить на CPU/RAM

За счёт кастомной реализации RPC-протокола на базе UDP мы ушли от проблем:

- head of line blocking
- растущего хвоста latency
- лишне потребляемых ресурсов

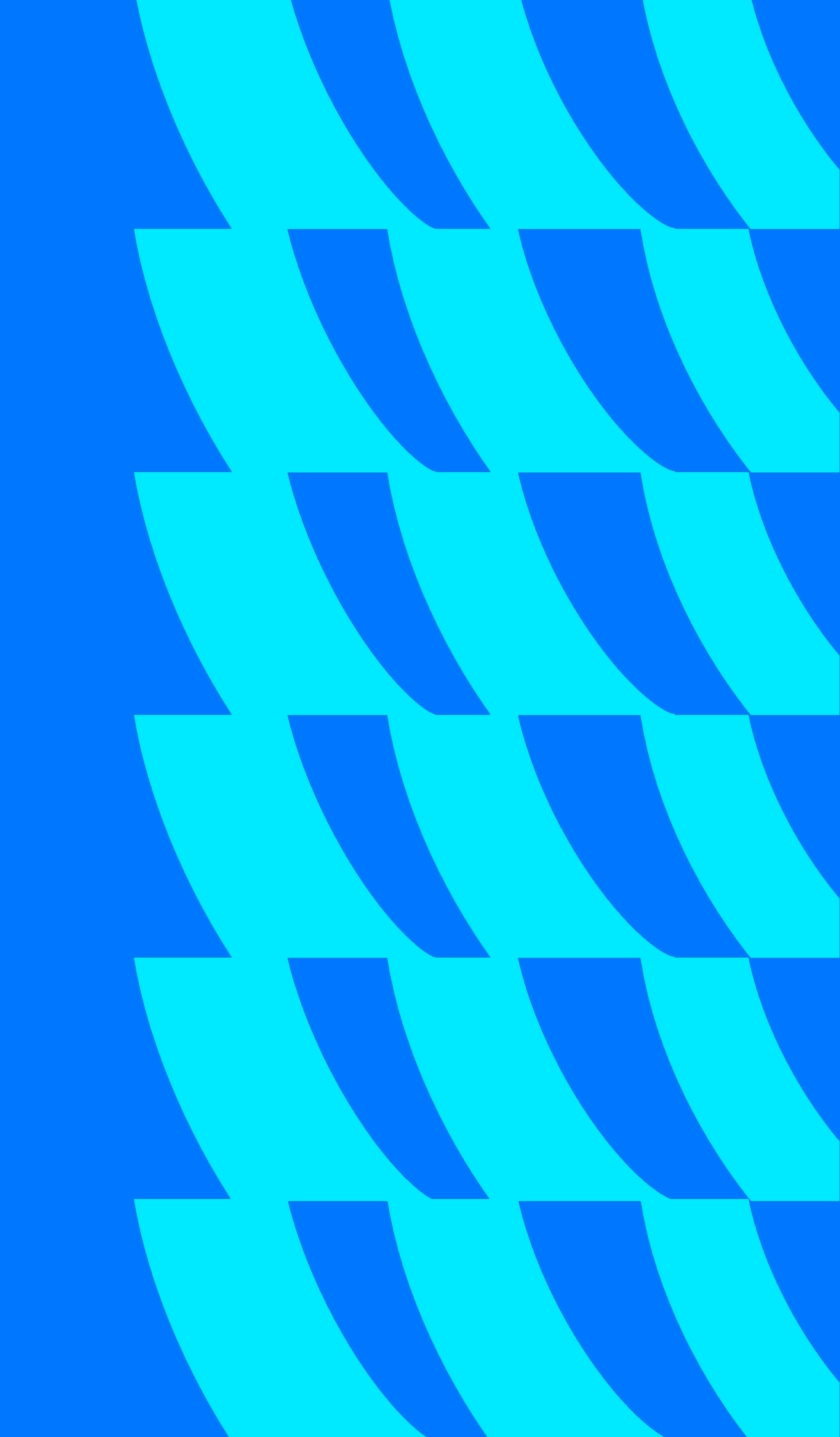
Когда записи немного, стоит задуматься об RCU

Мы научились не копировать
всю структуру данных и отказались
от счетчиков ссылок для сущностей

Но об этом в другом докладе ;)



Вместо заключения





Бенчмарк-бенчмарк-бенчмарк, не доверяйте экспертам



Используйте фазеры и property-based-тестирование



Будьте аккуратны с продакшном



Ошибаются все, главное – правильно сделать выводы



Спасибо за внимание!